AutomatedQA Corporation

AQtime 6

eliminate leaks and bottlenecks

User Manual

Copyright Notice

AQtime, as described in this on-line help system, is licensed under the software license agreement distributed with the product. The software may be used or copied only in accordance with the terms of its license.

© 2010 AutomatedQA Corporation. All rights reserved.

No part of this help can be reproduced, stored in any retrieval system, copied or modified, transmitted in any form or by any means electronic or mechanical, including photocopying and recording for purposes others than the purchaser's personal use.

All AutomatedQA product names are trademarks or registered trademarks of AutomatedQA Corporation. All other trademarks, service marks and trade names mentioned in this Help system or elsewhere in the AQtime software package are the property of their respective owners.

AQtime includes the UnzDll.dll library that is supplied by Info-Zip. This library is copyright © 1990-2005 Info-ZIP. This software is provided "as is", without warranty of any kind, expressed or implied. In no event shall Info-ZIP or its contributors be held liable for any direct, indirect, incidental, special or consequential damages arising from the use of or inability to use this software.

Table of Contents

INTRODUCTION **10**

AQtime Overview 10
AQtime x86 and x64 Packages 13
System Requirements 13
Supported Development Tools 15
Integration into IDEs 16
Supported Processor Models 19
What's New in AQtime 6 20
Features Added to AQtime 6.40 ..... *20*
Features Added to AQtime 6.30 ..... *20*
Features Added to AQtime 6.21 ..... *20*
Features Added to AQtime 6.20 ..... *20*
Features Added to AQtime 6.11 ..... *21*
Features Added to AQtime 6.10 ..... *21*
Features Added to AQtime 6.0 ..... *21*
Technical Support and Resources 23

GETTING STARTED **25**

User Interface Overview 25
AQtime Panels 33
AQtime Profilers 38
Preparing an Application for Profiling 43
How AQtime Profilers Use Metadata and Debug Information *43*
Compiler Settings for ASP.NET *45*
Compiler Settings for Microsoft Visual C# 2005 and 2008 ..... *45*
Compiler Settings for Microsoft Visual C# .NET *48*
Compiler Settings for Microsoft Visual J# 2005 and 2008 ..... *49*
Compiler Settings for Microsoft Visual J# .NET *52*
Compiler Settings for Microsoft Visual Basic 2005 and 2008 ..... *53*
Compiler Settings for Microsoft Visual Basic .NET *57*
Compiler Settings for Microsoft Visual C++ 2005 and 2008 ..... *58*
Compiler Settings for Microsoft Visual C++ .NET *61*
Compiler Settings for Borland C#Builder 2006 ..... *64*
Compiler Settings for Borland C#Builder *64*
Compiler Settings for Borland Delphi 2006 for .NET *65*
Compiler Settings for Borland Delphi 2005 ..... *66*
Compiler Settings for Borland Delphi 8 (Delphi for .NET) *67*
Compiler Settings for Microsoft Visual C++ 6.0 ..... *68*
Compiler Settings for Microsoft Visual Basic *71*
Compiler Settings for Embarcadero Delphi 2010 for Win32 ..... *72*
Compiler Settings for CodeGear Delphi 2009 for Win32 ..... *75*
Compiler Settings for CodeGear Delphi 2007 for Win32 ..... *78*
Compiler Settings for Borland Delphi 2006 for Win32 ..... *81*

Compiler Settings for Borland Delphi 3-7 ..... *84*
Compiler Settings for Embarcadero C++Builder 2010 ..... *86*
Compiler Settings for CodeGear C++Builder 2009 ..... *93*
Compiler Settings for CodeGear C++Builder 2007 ..... *98*
Compiler Settings for Borland C++Builder 2006 ..... *106*
Compiler Settings for Borland C++Builder 3-6 ..... *113*
Compiler Settings for Borland C++ *116*
Compiler Settings for Intel C++ *117*
Compiler Settings for GNU CC *117*
Compiler Settings for Compaq Visual Fortran *118*
Setting up the Project 120
Opening a Project *120*
Controlling What to Profile *124*
Excluding Code From Profiling *125*
Profiling Levels *126*
Defining Areas to Profile *127*
Checking Elements to Profile *131*
Using Triggers *132*
Setting Up Triggers *133*
Using Actions *135*
Selecting a Profiler *137*
Profiling and Analyzing Results 139
Doing One Profiler Run *139*
Analyzing Profiler Results *141*

AQTIME PANELS 146

Assistant Panel 146
Call Graph Panel 147
Call Graph Panel - Description *147*
Call Graph Panel Options *149*
Call Tree Panel 150
Call Tree Panel - Description *150*
Call Tree Panel Options *155*
Details Panel 156
Details Panel - Description *156*
Details Panel Options *157*
Disassembler Panel 157
Disassembler Panel - Description *157*
Disassembler Panel Options *160*
Editor Panel 162
Editor Panel - Description *162*
Editor Panel – General Settings *168*
Editor Panel – Display Settings *168*
Editor Panel – Highlighting Settings *169*
Editor Panel – User Keywords Settings *170*
Event View Panel 170
Event View Panel - Description *170*
Event View Panel Options *174*
Exceptions in the Event View Panel *177*
Adding Custom Messages to the Event View Panel *178*
Event View Panel – Possible Problems With the Call Stack *179*

Stack Frames – Compiler Settings *179*
Explorer Panel 180
Explorer Panel - Description *180*
Explorer Panel Options *182*
Monitor Panel 183
Monitor Panel - Description *183*
Monitor Panel Options *186*
Using the Monitor Panel With the Allocation Profiler *187*
PE Reader Panel 190
Report Panel 194
Report Panel - Description *194*
Report Panel Options *196*
Setup Panel 197
Setup Panel - Description *197*
Setup Panel Options *200*
Summary Panel 201

AQTIME PROFILERS 203

Performance Profiler 203
Performance Profiler - Overview *203*
Analyzing Performance Profiler Results *205*
Calculating Percent in the Report Panel *212*
<JIT compiler> and <Garbage collector> Routines *213*
<Root> Routine *215*
Performance Profiler - Report Panel Columns *215*
Performance Profiler - Details Panel Columns *224*
Performance Profiler Options *239*
Searching for Bottleneck Reasons Using the Performance Profiler *240*
Allocation Profiler 242
Allocation Profiler – Overview *242*
Allocation Profiler – Analyzing Visual C++ Applications *245*
Allocation Profiler – Analyzing Visual Basic 6.0 Applications *247*
Allocation Profiler – Analyzing Delphi Applications *248*
Allocation Profiler – Analyzing C++Builder Applications *250*
Allocation Profiler – Analyzing Intel C++, Borland C++ and GNU CC Applications *252*
Tracing System Management Memory Functions *253*
Tracing Attempts to Access Released Memory *253*
Checking Bounds of Memory Blocks *254*
Analyzing Allocation Profiler Results *256*
Allocation Profiler – Report Panel Columns *265*
Allocation Profiler – Columns of the Details and Call Tree Panels *267*
Allocation Profiler Options *268*
BDE SQL Profiler 270
BDE SQL Profiler - Overview *270*
BDE SQL Profiler - Report Panel Columns *272*
BDE SQL Profiler - Details Panel Columns *273*
BDE SQL Profiler Options *273*
Coverage Profiler 274
Coverage Profiler - Overview *274*
Coverage Profiler - Report Panel Columns *282*
Coverage Profiler - Details Panel Columns *284*

© 2010 AutomatedQA Corp.

Coverage Profiler Options *285*
Exception Trace Profiler 286
Function Trace Profiler 286
Function Trace Profiler – Overview *286*
Function Trace Profiler – Tracing Function Call Parameters and Result Values *290*
Function Trace Profiler – Report Panel Columns *291*
Function Trace Profiler – Details Panel Columns *295*
Function Trace Profiler – Outputting Results Using a Custom DLL *297*
Function Trace Profiler Options *298*
Light Coverage Profiler 300
Light Coverage Profiler – Overview *300*
Light Coverage Profiler - Report Panel Columns *306*
Light Coverage Profiler - Details Panel Columns *308*
Light Coverage Profiler Options *308*
Load Library Tracer 309
Load Library Tracer – Overview *309*
Load Library Tracer – Report Panel Columns *313*
Load Library Tracer – Details Panel Columns *314*
Load Library Tracer Options *315*
Reference Count Profiler 316
Reference Count Profiler – Overview *316*
Reference Count Profiler - Report Panel Columns *319*
Reference Count Profiler - Columns of the Details and Call Tree Panels *320*
Reference Count Profiler Options *321*
Platform Compliance Profiler 322
Platform Compliance Profiler - Overview *322*
Platform Compliance Profiler - Report Panel Columns *325*
Platform Compliance Profiler Options *326*
Resource Profiler 326
Resource Profiler - Overview *326*
Analyzing Resource Profiler Results *327*
Resource Profiler - Report Panel Columns *333*
Resource Profiler - Details Panel Columns *335*
Non-Existent Resources in the Report Panel *335*
Resource Profiler Options *336*
Resource Profiler - List of Checked Functions *337*
Sequence Diagram Link Profiler 347
Sequence Diagram Link Profiler - Overview *347*
Sequence Diagram Link Profiler Options *350*
Static Analysis Profiler 350
Static Analysis Profiler - Overview *350*
Static Analysis Profiler - Analyzing Results *362*
Static Analysis Profiler - Report Panel Columns *372*
Static Analysis Profiler - Columns of the Details and Call Tree Panels *374*
Static Analysis Profiler Options *376*
Unused VCL Units Profiler 376
Unused VCL Units Profiler - Overview *376*
Unused VCL Units Profiler - Report Panel Columns *379*
Unused VCL Units Profiler - Details Panel Columns *379*
Unused VCL Units Profiler Options *380*
Counters Overview 380

HOW TO **387**

Common Tasks 387
Disabling inlining for the managed application to be profiled *388*
Finding memory block violations *388*
Finding routines exported and imported by a module *388*
Finding the routine where an exception occurred *389*
Finding where a method or a class is defined in source code *389*
Knowing average, maximum and minimum execution times for a method *389*
Knowing if a method raised exceptions *390*
Knowing on which platforms your application can run *390*
Knowing parameters and result values of function calls *390*
Knowing the number of clients that refer to an interface object *390*
Knowing the number of entries into a method *391*
Knowing the structure of potential interlinks between classes in your application *391*
Knowing the structure of references between objects in your application *391*
Knowing the structure of routine calls in your application *391*
Knowing the total time spent on a method (excluding child methods) *392*
Knowing the total time spent executing a method (including child methods) *392*
Knowing what binary or MSIL code a method has *392*
Knowing what libraries your application uses *392*
Knowing what methods are called the most or the least often *393*
Knowing what methods take up the most or the least execution time *393*
Knowing what methods use the most time for JIT compiling *394*
Knowing what methods were executed *394*
Knowing what source code lines were executed *395*
Knowing what source code lines are called the most or the least often *395*
Knowing what source code lines take up the most or the least execution time *395*
Searching for memory leaks *396*
Searching for resource leaks and errors in resource management functions *397*
Tracing references between objects *398*
Profiling Applications 398
Optimizing the Profiling Process *398*
Getting Results During Testing *399*
Clearing Results During Profiling *401*
Profiling Modes *401*
"Attach to Process" *402*
Profiling Under 64-bit Platforms *404*
Profiling Dynamic Link Libraries *405*
Profiling Web Server Applications *407*
Profiling ASP.NET Applications *408*
Profiling ASP.NET Applications via ASP.NET Development Server *410*
Profiling ASP.NET Applications via Internet Information Services *413*
Profiling IIS Applications *421*
Profiling ASP.NET Applications: ASP.NET or IIS is Running Under a User Account *430*
Profiling COM Applications *431*
Profiling In-Process COM Servers *432*
Profiling Out-of-Process COM Servers *433*
Profiling DCOM Servers *434*
Profiling COM+ and MTS Applications *435*
Profiling COM Logical Threads *438*

Profiling Scripts *439*
Profiling Scripts – Prerequisites *441*
Profiling Scripts Using Host Applications *445*
Profiling Script Files *445*
Profiling Scripts Located on Web Pages *447*
Profiling Scripts in Internet Explorer 8 ..... *448*
Profiling Scripts – Troubleshooting *449*
Profiling Services *450*
Profiling SQL Server CLR Integration Assemblies *451*
Profiling WPF Browser (XBAP) Applications *457*
Profiling Multiple Threads *459*
Profiling Multiple Processes *462*
Profiling System Calls *464*
Profiling Managed and Unmanaged Code *466*
Profiling Recursive Routines *467*
Profiling Duplicated Code *469*
Profiling Small Functions *470*
Profiling Inline Functions *471*
Profiling Template Functions *474*
Profiling Routines That Hold Unsafe Code *474*
Profiling Routines That Do Not Have the ret Instruction *476*
Profiling Child Routines Along With Parents *477*
Profiling Startup Code *478*
Controlling Profiling From Application Code *481*
Generating Dumps for Profiled Applications *482*
Assigning Names to Threads *483*
Specifying Path to Debug Info Files *485*
Profiling .NET Applications – Peculiarities *486*
Adding Code to Profiling Areas From Code Editor *486*
Working With Profiler Results 487
Adding Selected Routines and Classes to the Setup Panel *487*
Comparing and Merging Results *488*
Sorting Results *492*
Grouping Results *492*
Searching Results *493*
Filtering Results *494*
Result Views *495*
Printing Profiler Results *496*
Exporting Results *497*
Structure of XML Results *498*
Exporting Profiling Results to Database *499*
Working with AQtime via COM 500
Working with AQtime via COM - Overview *500*
Working With AQtime From Managed Code *504*
Methods and Properties of the IntegrationManager Object *505*
Getting and Settings Profiling Mode Parameters *509*
Exporting Results to Database via COM *510*
Miscellaneous 511
Integration With Microsoft Visual Studio Team System *511*
AQtime Command Line *515*
Working With Source Control Systems *516*

TestComplete Information *518*
AQtrace Information *519*
AQtime Data Files *520*
INDEX **521**

Introduction

AQtime Overview

From specification to final delivery, professional developers constantly aim to build applications that are robust, clean-running and clear of hidden bottlenecks, resource wastage and performance limitations. AQtime is the tool that tells you at any moment during development how your application is doing. Using its comprehensive suite of profilers, developers can measure the health of their applications with unrivaled accuracy.

Profiling vs. testing

AQtime is an **integrated profiling toolkit**. You may be asking yourself what the difference is between a profiler and a regression testing tool? A test tool records what each part of an application does for other parts, and **what** the entire application does for the user. A profiler traces **how** the application does what it does. A test tool takes **output** measurements. In essence, a profiler takes "**health & vitality**" measurements. Needless to say, AutomatedQA makes an excellent test tool, TestComplete. But now that we have shown that profiling and testing are two different things, the rest of this documentation will be concerned only with what AQtime does, profiling.

Manual vs. automation

You can certainly profile an application manually, without AQtime. Manual profiling in its most rudimentary form might be to use a stopwatch and system tools to check resources before and after application usage. A more advanced method of manual profiling is to insert code within your application and check the system timer at the start and at the end of a code section, check resources, output routine calls to a log, etc. In fact, without a comprehensive automated profiler like AQtime, this is what you will be forced to do when you are worried about the "health & vitality" of a section of code.

A profiler, on the other hand, **tracks** and accurately **measures** performance and memory use during application execution automatically, then displays the results in **comparative** format. For instance, it might time the start and end of any routine call, and display the results as a percent of total time used by each routine.

What AQtime does

AQtime removes the dangerous guesswork traditionally associated with performance and memory usage optimization. It offers you an easy-to-use and structured way to hunt down and eliminate the cause of bottlenecks as well as memory hogging unsafe code. With AQtime you can be on your way to building applications that perform at their highest possible level - all the time!

AQtime was built with a single key objective - to help you completely understand how your programs perform during execution. AQtime gathers crucial performance and memory allocation information at runtime and delivers it to you in both summarized and detailed form, with all the tools you will need to begin the optimization process - from customized filters and graphical call hierarchies to source code views.

Of course, taking only one kind of profile is not a way of keeping on top of the general health of the application. This is like tracking your health with a balance. A good automatic profiling application will supply

several kinds of profilers, and allow you to use any number together or separately, and on varied parts of the application. Better yet, it will let you interactively pin down the crucial information you are looking for, and which may not be where you thought it would be at first.

Not only can a profiler take many kinds of measurements (how often a function is called, time spent in a given unit, events generated, memory leaks, etc.), it can also get these in different ways:

- Some of it is totally non-intrusive: the profiler requests before-and-after information from the operating system.
- Some of it is practically non-intrusive: modern operating systems switch tasks many times per second. On each task switch, which would equally occur without the profiler being present, the profiler can gather some extremely simple information; what this changes to the task switch is immeasurable; the profiler's only practical intrusion is that it uses some memory and resources.
- Some of it is minimally intrusive: profiling operations are inserted at many spots, but they are inserted through binary instrumentation. That is, once the executable code is loaded into memory, it is modified to add the needed operations. This is better than source-code instrumentation not only for the reasons explained below, but because in binary the profiling points can be positioned more precisely. For instance, a short (but often-called) function may spend most of its time in setup and finalization, that is, before the first line of code and after the last. Instrumenting source cannot profile those parts of it, so it yields highly misleading information.
- Some of it is awkwardly intrusive. The processor allows a soft breakpoint operation, which in principle would be the simplest way to call profiler services. So, one variation of binary instrumentation or source code modification (see below) is to insert these "made to order" soft-breakpoint instructions. Under Windows NT or 2000, however, each soft breakpoint implies a context switch, as the profiler is running as a separate process. The result is that most of the runtime will be occupied by these context switches.
- Some of it highly intrusive: the profiler requires modification of the source code, so that your profiling source is never your normal-build source. Since this implies thousands of insertions, it has to be done by an automated source-modification tool. The tool will tempt you into "avoiding" the forking by letting it undo the modifications it did. This is worse still - you can never be sure the "cleaned up" source is identical to what you had in the first place. Some people have sworn off automated profilers because of these intrusions.

As you might expect by now, AQtime **never**, ever **modifies source code**. In fact, **it always uses the least intrusive method** to achieve the requested results. However, since you normally expect results to refer to functions or sections of code, most AQtime profiles require that the application be compiled with debugger information, so that code points can be linked to function or unit names.

In addition, AQtime does not use soft breakpoints, with their context switches. The operations added by binary instrumentation are minimal, and run in the same process as the application. It should be noted, however, that binary instrumentation is still instrumentation. The operation may be very quick, but it leaves the processor, with its pipelines and caches, in a somewhat different state than if there had been no instrumentation.

A note about results: many profiles are measures of relative time. In the ordinary world, relative time is time relative to total elapsed time, that is, real time. In profiling generally, it's different. You cannot do anything about the elapsed time spent waiting for user input, except to go without the input. You can somewhat easier go without some system calls, but the fact remains that you cannot improve system code. Therefore, a profiler by default compares profiled times - the times your own code takes to execute. Relative time is time relative to the time taken by all profiled functions.

AQtime, of course, allows you to get profiles for each function taken alone, or including all the calls it makes to other functions ("child calls"). It also allows you to **include** or **exclude** time spent calling the **system**

functions. And finally it allows you to profile functions not just relative to one another, as is the usual practice for profilers, but relative to real time, the entire elapsed time of the profile run, that is, including input and output calls.

A hidden but crucial aspect of AQtime is that its architecture is COM-based. This means it can be used as a server by any application (the idl is supplied of course). In fact, it is used for some services (for example, coverage) by AutomatedQA's test automation software, *TestComplete*. More important is that all the parts of AQtime are COM objects. They can be separately plugged in or out. In fact, some of the profilers we will list below are supplied with your installation as separate plug-ins. More will be made available, or are already available on AutomatedQA's Web site, http://www.automatedqa.com/.

Therefore, each of these profilers is a standalone object; each is built and tuned to its one purpose. We are not talking about surface "features" added to the same basic engine, we are talking about separate, professional-grade profilers. The business of making them easy to understand and run, and of integrating their results together in a flexible format, is left to the User Interface, discussed further down.

The current list of profilers is in a separate topic, *AQtime Profilers*. You should read that before proceeding - it is the heart of AQtime.

The User Interface's main tasks are to allow you to:

- Specify the application to profile (project).
- Choose profilers to run on it.
- Filter the profiler results to center on your particular points of concern.
- Display the results.
- Format reports.

Results can be filtered by time, location, etc. They can also be filtered by the thread in which the event occurred.

There are many display options. Most results can be shown in one or several graphical formats (e.g. histogram), or in a selection of columns.

One display mode for the profiler results deserves special attention: the Call Graph. All binary-instrumented profilers in AQtime can record the caller for each call of a function. The problem is what to do with the resulting data. The Call Graph is a very easily understood, interactive display that shows each profiled function with basic timings, and arrows from the functions that called it, and to those that it called. Each arrow carries the count of calls recorded.

The most controllable form of result display is the report, which is a totally-configurable grid shown in the Report panel. Besides this onscreen display, the Report panel can print its contents and export it to a text, Excel, html or xml files.

Integration into IDEs

AQtime is tightly integrated into Microsoft Visual Studio and Borland Developer Studio. This feature provides you with the full AQtime functionality without leaving the IDE. You can create and manage AQtime projects, profile your applications and view profiling results directly from the IDE. See *Integration into IDEs* for more information.

Getting Started

To start using AQtime, see the *Getting Started* section of this document.

AQtime x86 and x64 Packages

AQtime includes two packages: AQtime x86 and AQtime x64. The difference is in the type of applications they can profile and in supported operating systems.

The **AQtime x86** package contains modules, files and componenets for profiling 32-bit Windows and .NET applications. Use this package to profile 32-bit applications on 32-bit Windows operating systems. Note that AQtime x86 can also run on 64-bit versions of Windows. However in this case, the Performance and Function Trace profilers can use only the *Elapsed Time* counter. The other counters are not available.

The **AQtime x64** package contains specific modules and components that are used to profile both 32- and 64-bit applications. Use this package to profile 32-bit and 64-bit Windows and .NET modules on a 64-bit Windows operating system (this package can run only on a 64-bit OS). Note that the Performance and Function Trace profilers can use only the *Elapsed Time* counter. The other counters are available if the operating system is running in debug mode. For detailed information on peculiarities of running AQtime x64, see *Profiling Under 64-bit Platforms*.

System Requirements

Hardware Requirements

- Intel Pentium II 450 or higher (Pentium 4 1GHz recommended). AQtime also supports the following processors:
  - Processors of the Intel Core family (Intel Core i7, Intel Core 2 Duo, Intel Core Duo and others)
  - Intel Xeon and Intel Xeon MP
  - Intel Pentium II, Intel Pentium III
  - Intel Pentium 4 (including Intel Pentium 4 supporting Hyper-threading Technology and Intel Pentium 4 Extreme Edition supporting Hyper-Threading Technology)
  - Mobile Intel Pentium 4
  - Intel Pentium Extreme Edition, Intel Pentium D, Intel Pentium M
  - Intel Celeron, Intel Celeron D, Intel Celeron M
  - AMD Phenom
  - AMD Athlon 64 FX
  - AMD Athlon XP, AMD Athlon 64 X2 Dual-Core, AMD Athlon 64
  - AMD Sempron
  - AMD Opteron
  - AMD Turion 64 X2 Mobile Technology and AMD Turion 64 Mobile Technology
  - AMD Athlon 64 for DTR
  - Mobile AMD Athlon 64
  - Mobile AMD Sempron

  To learn AQtime's limitations that depend on the engaged processor model, see *Supported Processor Models*.
- 256MB of RAM (1GB or more recommended).
- 250MB hard disk space for installation and 200MB for using the product.

© 2010 AutomatedQA Corp.

- SVGA (800 × 600) or higher resolution monitor.
- Mouse or other pointing device.

Note: AQtime may consume a lot of memory to store profiler information. Therefore, when working with large projects, it is recommended that you allocate as much physical RAM as possible so that Windows does not use the swap file.

Operating System and Internet Explorer

- Operating system:
  - Profiling of .NET applications requires Microsoft Windows 7 (either 32-, or 64-bit edition), Microsoft Windows Server 2008 (either 32-, or 64-bit edition), Microsoft Windows Vista (either 32-, or 64-bit edition), Windows XP (either 32-, or 64-bit edition), Windows Server 2003 (either 32-, or 64-bit edition), Windows 2000, or Windows NT 4.0 Service Pack 6 with Microsoft .NET Framework 1.0, 1.1, 2.0, 3.0 or 3.5.
  - Profiling of native-code (non-.NET) applications requires Microsoft Windows 7 (either 32-, or 64-bit edition), Microsoft Windows Server 2008 (either 32-, or 64-bit edition), Microsoft Windows Vista (either 32-, or 64-bit edition), Windows XP (either 32-, or 64-bit edition), Windows Server 2003 (either 32-, or 64-bit edition), Windows 2000, or Windows NT 4.0 with Service Pack 6.
- Microsoft Internet Explorer 5.0 or later.

Important notes:

- If you use a computer that has several processors or a multiple-core processor (for example, dual-core CPU) and has Windows XP Service Pack 2, then you **must install the Windows update #896256** in order for AQtime to be able to profile your application correctly. The update is available on Microsoft's web site:
  - http://support.microsoft.com/kb/896256
- In order for AQtime to be able to function on Windows Server 2008 R2, the WoW64 component is required. The Server Core installation option for Windows Server 2008 R2 does not install WoW64 by default, so, you should install it yourself. This requirement concerns both AQtime x86 and x64 packages.

Profiling Scripts

To profile VBScript and JScript functions, AQtime requires the Windows Script and Script Debugger components. The Windows Script component is supplied with each Microsoft operating system (since Windows 98), thus it is already present on your machine. The Script Debugger may also be installed on your machine, as it is shipped along with Visual Studio (since version 2003) and Microsoft Office (version XP and later).

If you have problems with script profiling, try reinstalling these components. You can find standalone packages of these components at Microsoft Download Center (http://www.microsoft.com/downloads). Search for the following items:

- Windows Script (Windows Script 5.6 or later is required)
- Script Debugger

The versions of Windows Script and Script Debugger may be different and incompatible with one another. For information on possible problems and workarounds for them, see *Profiling Under 64-bit Platforms*.

To profile script routines, you also need to specify certain user permissions and prepare the host application in a certain way. See *Profiling Scripts - Prerequisites* for more information.

User Account

To install and use AQtime, you must have administrator permissions on your computer. AQtime can be installed and used under different user accounts, but all of them must have administrator permissions.

Running Under Windows Vista, Windows Server 2008 or Windows 7

To be used under Windows Vista, Windows Server 2008 or Windows 7, AQtime must be launched under an account with administrator privileges.

AQtime and Windows DDK

If you have Windows DDK installed, then after installing AQtime:

- Launch Driver Verifier (a tool from the Windows DDK package) and
- Disable verification of the aqIPD.sys driver (the aqIPD.sys driver is part of AQtime).

Driver Verifier blocks the AQtime driver, so you need to disable verification for the AQtime driver.

Integration into IDEs

AQtime can be integrated into Microsoft Visual Studio and Embarcadero RAD Studio (and into earlier versions of RAD Studio by CodeGear and Borland):

- Microsoft Visual Studio 2008, Microsoft Visual Studio 2005,
  Microsoft Visual Studio .NET 2003, Microsoft Visual Studio .NET.
- Embarcadero RAD Studio 2010.
- CodeGear RAD Studio 2007 and 2009.
- AQtime can be integrated into Borland Developer Studio 2006.

  Integration will be possible only if Borland Developer Studio 2006 Update 2 is installed on your computer. Integration into localized (non-Elnglish) versions is currently not supported.

Supported Development Tools

AQtime can profile both .NET (managed) and native-code (non-.NET, unmanaged) executables.

.NET (Managed) Applications

AQtime supports all existing compilers that generate MSIL code, for example:

Microsoft Visual Studio .NET 2002 and 2003, Microsoft Visual Studio 2005 and 2008	Outside Microsoft
Visual C# 2005, 2008	Embarcadero RAD Studio 2010
Visual C# .NET	CodeGear RAD Studio 2007 and 2009
Visual Basic 2005, 2008	Borland Delphi 2006 for .NET
Visual Basic .NET	Borland Delphi 2005 for .NET
Visual C++ 2005, 2008	Borland Delphi 8 (Delphi for .NET)
Visual C++ .NET	Borland C#Builder 2006

Visual J# 2005, 2008	Borland C#Builder
Visual J# .NET	APL
JScript .NET	Cobol
F#, Visual F#	Component Pascal
	Eiffel
	Haskell
	Mercury
	Oberon
	Perl
	Python
	Scheme
	SmallTalk
	Standard ML

This list is expanding constantly while new .NET-friendly languages appear.

To learn more about Microsoft .NET, visit http://www.microsoft.com/net/.

Native-Code (Unmanaged) Applications

AQtime can profile executables created with any of the following development tools:

- Microsoft Visual C++ v. 4, 5, 6, 7, 8 and 9
- Microsoft Visual Basic v. 6.0
- Embarcadero Delphi 2010, CodeGear Delphi 2007 and 2009 for Win32, Borland Delphi 2050 and 2006 for Win32, Borland Delphi v. 2, 3, 4, 5, 6, 7
- Embarcadero C++Builder 2010, CodeGear C++Builder 2007 and 2009, Borland C++Builder 2006, Borland C++Builder v. 3, 4, 5, 6
- Intel C++ v. 7.0
- Borland C++ v. 4.5 and 5.x
- GNU Compiler Collection v .2.95 and later
- Compaq Visual Fortran v. 6.5

Besides support for compilers included in Microsoft Visual Studio and Embarcadero RAD Studio (and earlier versions of RAD Studio by CodeGear and Borland), AQtime is tightly integrated into these IDEs. For more information, see *Integration into IDEs*.

Integration into IDEs

AQtime can be tightly integrated into Microsoft Visual Studio and Embarcadero RAD Studio (or into earlier versions of RAD Studio by CodeGear and Borland). The integration provides developers with full control over AQtime without leaving the IDE, ensuring a continuous application development experience. Leveraging the tremendous power of AQtime's unparalleled application analysis tools is now as easy as debugging from the IDE.

Besides core integration into the abovementioned IDEs, it is possible to add an AQtime menu item to other development tools. Using this menu item, you can quickly create an AQtime project for the current application.

Integration into Microsoft Visual Studio

AQtime can be tightly integrated in the Microsoft Visual Studio IDE. Currently, AQtime extends Microsoft Visual Studio .NET 2002, Visual Studio .NET 2003, Visual Studio 2005 and Visual Studio 2008.

The integration of AQtime into Visual Studio means:

- **AQtime panels (Setup, Report, Summary and others) are integrated into Visual Studio's IDE**. When you open an AQtime project in Visual Studio, the layout of the Visual Studio panels and windows is extended with AQtime-specific panels. When an AQtime project is closed, these panels are automatically hidden within Visual Studio.
- **A new AQtime project type is added**. You can create AQtime projects or add them to an existing Visual Studio solution the same way that you create or add other VS projects; by using the **Create Project** and **Add New Project** dialogs of Visual Studio. The contents of an AQtime project are displayed in the **Solution Explorer**.
- **AQtime integrates its menus and toolbars into the Visual Studio IDE**. AQtime adds the following menu items and toolbars to Visual Studio's menu and toolbar system:
  - AQtime adds the **Profile** menu to Visual Studio's main menu. This menu contains commands that let you start, pause and resume profiling, as well as select the profiling mode, modify the profiler and panel options, and so on.
  - Project-specific commands, such as **Add Module**, **Add Output** or **Add Assembly**, are added to the **Project** menu, and are also available through the context menu of AQtime's project node in the Solution Explorer.
  - AQtime also adds the **AQtime** toolbar to Visual Studio. This toolbar contains the most frequently used items, such as **Run**, **Select Profiler**, and so on.

  Also, most of the standard commands (**Select All**, **Delete**, **Print**, **Find**, and others) are applicable to AQtime panels.
- **AQtime contributes to the Automation Model**. The automation model of Visual Studio provides users with the ability to create custom add-ins, wizards, and work with macros. As a true Visual Studio-integrated product, AQtime provides program interfaces for its internals (for example, for the Projects interface). This allows third-party developers to create Visual Studio add-ins, wizards and macros that use AQtime's object model.
- **AQtime integrates into the Visual Studio Help system**. F1 context-sensitive Help is provided for all AQtime panels, dialogs and project items. The **Dynamic Help** window provides a list of help topics specific to the current AQtime context or task you are currently working on.
- **Further integration with Visual Studio Studio**. In addition to the core integration, AQtime supports advanced integration that displays product information in the About dialog box and on the splash screen, Properties window support, integration into Visual Studio's Code Editor and Start Page, and more.
- **AQtime integrates into Microsoft Visual Studio 2005 Team System**. AQtime also includes a special package for integrating Microsoft Visual Studio 2005 Team System. This package gives Quality Assurance engineers and testers the possibility to extend Visual Studio test projects with AQtime projects. Test projects created in Visual Studio run AQtime to search for memory or resource leaks in the application or to check the platform compliance. This feature makes your testing even more flexible and powerful. For more information, see *Integration With Microsoft Visual Studio 2005 Team System*.

Integration into Embarcadero RAD Studio

AQtime can be integrated into Embarcadero RAD Studio and earlier versions of this product produced by CodeGear and Borland. Currently, AQtime extends the following versions of these IDEs:

- Embarcadero RAD Studio 2010
- CodeGear RAD Studio 2007 and 2009
- Borland Developer Studio 2006

  Note: Integration is only possible if Borland Developer Studio 2006 Update 2 is installed on your computer.

Note: Intergation into localized (non-English) versions of Borland Developer Studio or CodeGear RAD Studio is not supported at the moment.

AQtime's integration into RAD Studio IDE means the following:

- **AQtime panels (Setup, Report, Summary and others) are integrated into the IDE**. When you open an AQtime project in RAD Studio, the layout of the IDE's panels and windows is extended with AQtime-specific panels. When an AQtime project is closed, these panels are automatically hidden within the IDE.
- **A new AQtime project type is added**. In RAD Studio, AQtime projects (.aqt files) are part of the AQtime project groups (.bdsproj files). You can create AQtime projects and project groups the same way you create Delphi projects and project groups of other types by using the **New Items** dialog of RAD Studio. The contents of an AQtime project are displayed in the Project Manager.
- **AQtime integrates its menus and toolbars into the IDE**. AQtime adds the following menu items and toolbars to the IDE's menu and toolbar system:
  - AQtime adds the **Profile** menu to the IDE's main menu. This menu contains commands used to choose the profiler, modify the profiler and panel options, and so on.
  - AQtime inserts the **Run With Profiling** item into the **Run** menu.
  - AQtime toolbars are added to RAD Studio and let you perform other AQtime-specific actions in the IDE.
- **AQtime integrates into the RAD Studio Help system**. F1 context-sensitive Help is provided for all AQtime panels, dialogs and project items.
- **Further integration into RAD Studio**. In addition to the core integration, AQtime supports advanced integration: information about the product is displayed in the About dialog box and on the splash screen, AQtime can be integrated into the IDE's Code Editor, and more.

Adding the AQtime Menu Item to IDE

For easy access, the AQtime item can be added to the Tools menu in Borland Delphi, C++Builder and Microsoft Visual C++ 6.0 IDE. Once this is done, selecting AutomatedQA AQtime 5 from the Tools menu in the development tool will launch AQtime, compile the current application with debug information and load it in AQtime. This functionality is supported for applications that create executables, it is not availabe for DLLs.

The AQtime installation program automatically adds the AutomatedQA AQtime 5 menu item to the Tools menu. For instructions on how to do this manually, see the following topics in on-line help:

- *Adding the AQtime Menu Item to Borland Delphi and C++Builder IDE*
- *Adding the AQtime Menu Item to Visual C++ 6.0 IDE*

Supported Processor Models

As the *System Requirements* topic states, AQtime can operate on computers that include any of the following processor models if the processor provides appropriate performance:

- **Intel Processors**
  - Processors of the Intel Core family (Intel Core i7, Intel Core 2 Duo, Intel Core Duo and others)
  - Intel Xeon and Intel Xeon MP

    Some counters are not supported on the Intel Xeon multi-core processors with the Hyper-Threading technology, for instance, on Intel Xeon Duo Core processors with hyper threading. See below).
  - Intel Pentium II
  - Intel Pentium III
  - Intel Pentium 4 (including Intel Pentium 4 supporting Hyper-threading Technology and Intel Pentium 4 Extreme Edition supporting Hyper-Threading Technology)
  - Mobile Intel Pentium 4
  - Intel Pentium Extreme Edition
  - Intel Pentium D
  - Intel Pentium M
  - Intel Celeron
  - Intel Celeron D
  - Intel Celeron M
- **AMD Processors**
  - AMD Phenom
  - AMD Athlon 64 FX
  - AMD Athlon 64 X2 Dual-Core
  - AMD Athlon 64
  - AMD Sempron
  - AMD Opteron
  - AMD Athlon XP
  - AMD Turion 64 X2 Mobile Technology
  - AMD Turion 64 Mobile Technology
  - AMD Athlon 64 for DTR
  - Mobile AMD Athlon 64
  - Mobile AMD Sempron

Using some of the processor models mentioned above imposes certain limitations on AQtime's functionality. These limitations mean that particular profiler counters of the **Performance** profiler are not available for these “exclusive” processor models. The Intel Pentium 4 and Intel Pentium D processors are free of these limitations. They support all counters. The currently known limitations of other processors are as follows:

- The **Intel Core i7**, **Intel Core 2 Duo**, **Intel Pentium II**, **Intel Pentium III**, **Intel Pentium M**, **AMD Phenom**, **AMD Athlon XP** and **AMP Athlon 64** processors support the *Elapsed Time*, *User Time*, *User+Kernel Time*, *CPU Cache Misses*, *CPU Mispredicted Branches*, *Context*

Switches, *Hard Memory Page Faults*, *Soft Memory Page Faults* and *All Memory Page Faults* profiler counters but do not support the *Split Load Replays*, *Split Store Replays*, *Blocked Store Forwards Replays* and *64K Aliasing Conflicts* counters.

- The **Mobile Intel Pentium 4** processor and the **AMD Opteron** and **AMD Turion** processors only support the *Elapsed Time*, *Context Switches*, *Hard Memory Page Faults*, *Soft memory Page Faults* and *All Memory Page Faults* counters.
- The **Intel Xeon** and **Intel Xeon MP** multi-core processors with the Hyper-Threading technology only support the *Elapsed Time*, *Context Switches*, *Hard Memory Page Faults*, *Soft Memory Page Faults* and *All Memory Page Faults* counters.

  The single-core **Intel Xeon** and **Intel Xeon MP** processors support all the counters.

Note: If you run AQtime x86 on a 64-bit operating system, the only available counter is *Elapsed Time*. The other counters are unavailable. AQtime x64 does not impose any limitations on counters. See *AQtime x86 and x64 Packages* for more information.

What's New in AQtime 6

AQtime 6 includes a vast amount of new features. Below is a complete list of changes for AQtime 6. For information about changes in previous versions of AQtime, see *Version History* in on-line help.

Features Added to AQtime 6.40

- Support for applications created with Embarcadero RAD Studio 2010 and integration into this version of the IDE.
- Support for AMD Phenom processors (see *Supported Processor Models*).
- The Platform Compliance profiler now supports Windows 7.
- A number of bugs have been fixed.

Features Added to AQtime 6.30

- Support for Microsoft Windows 7. Now you can profile your applications under this operating system.
- Now AQtime supports Intel Core i7 processors.
- A new script profiling mode that is optimized for Internet Explorer 8 and does not require a script debugger. Read *Profiling Scripts in Internet Explorer 8* for further details.
- A number of bugs have been fixed.

Features Added to AQtime 6.21

- A number of bugs have been fixed.

Features Added to AQtime 6.20

- The Event View panel has new toolbar buttons that help navigate through events that were logged for the same thread.

- A number of bugs have been fixed.

Features Added to AQtime 6.11

- A number of bugs have been fixed.

Features Added to AQtime 6.10

- Now AQtime can profile Delphi 2009 and C++Builder 2009 applications.
- The errors when profiling applications created with the Microsoft F# compiler were fixed.
- The code editor now highlights Microsoft F# syntax elements.
- A number of bugs have been fixed.

Features Added to AQtime 6.0

Overall Improvements

- **Profiling scripts.** Now you can use AQtime to profile scripts executed by the Microsoft Scripting Engine. These, for example, include VBScript and JScript routines located on web pages as well as TestComplete scripts. To profile a script, you should launch the host application under AQtime and then execute the script in that application. AQtime recognizes the script's activity and reports its results along with the application results. Currently, scripts can be profiled with the Performance, Coverage and Function Trace profilers. For more information about this, see *Profiling Scripts*.
- **Optimization of function parameters retrieval.** The Function Trace profiler has been optimized to increase its performance when gathering the values of function call parameters. In previous versions, these values were collected from the entire application if the Get parameter info profiler option was enabled. As a result, a lot of memory was required to collect the results, which slowed down the profiling process significantly.

  In AQtime ver. 6, this feature is in the profiling area settings. Now including areas have the Retrieve parameter values property. If the property is enabled, the Function Trace profiler only logs the actual parameter values for the functions that belong to the area. Thus, the profiler only inspects the routines you are interested in.
- **Filtering out standard units.** Most development environments include their standard libraries in applications. The library routines perform a lot of auxiliary tasks, interact with classes and controls, draw forms, and so on. However, generally, profiling of standard units does not make a lot of sense, since their code cannot be modified. To only focus on your code, you can exclude standard libraries from profiling. This functionality can be enabled via the *Exclude standard source files* option, or via the **Exclude Standard Source Files** button of the Setup panel.
- **Extended .NET code debugger.** AQtime can now obtain a call stack for .NET routines, trace specific .NET events and display this information in the Event View panel. This functionality is available if the *Use extended debugger for managed code* option is enabled.

New Profilers

- **New Reference Count Profiler.** AQtime includes a new Reference Count profiler in version 6. With its help you can trace references to interface objects that implement the `IUnknown` interface or its descendants. Analyzing the number of references and the way they are created and removed,

you can track unreleased or prematurely released interface objects. Read *Reference Count Profiler - Overview* for more information.

- **New BDE SQL Profiler.** Using a new BDE SQL profiler you can time the execution of stored procedures and queries made to the databases via the Borland Database Engine.
- **New Unused VCL Units Profiler.** This profiler is designed to decrease the size of Delphi applications. It analyzes the application's source code and debug information. Using the profiler you can detect both standard and user units that are added to the project but are not used by the application. These units can safely be excluded thus reducing the application's size. Read *Unused VCL Units Profiler - Overview* for more information.

Allocation and Resource Profiler Improvements

- **Allocation profiler support for release versions of Visual C++ applications.** In previous versions of AQtime, the Allocation profiler could only retrieve data from Visual C++ applications compiled in the Release configuration if the application was prepared in a special way. Now these preparations are not required. See *Allocation Profiler - Analyzing Visual C++ Applications* for details.
- **Tracing attempts to access released memory blocks.** The Allocation profiler has a new *Fill released memory blocks* option that overwrites the released memory with invalid data, so an attempt to read data from a released block will result in an error. Using this feature you can detect functions that read data from de-allocated memory blocks.
- **Changes in the algorithm of checking bounds of memory blocks.** The algorithm was improved to perform memory verification not only on block re-allocation or release, but also on retrieving results and on terminating the application. This helps to pin-point violations of memory block bounds even in for memory leaks or when the corresponding blocks were not released. Read *Checking Bounds of Memory Blocks With the Allocation Profiler* for more information.
- **GDI Plus support for the Resource profiler.** The Resource profiler contains a new *GDI+ Resources* category. Use this category to track the execution of GDI+ functions. See *Resource Profiler - "GDI+ Resources" Function Category (gdiplus.dll)*.

Other Improvements

- The data shown in the Call Tree panel can now be stored in a separate file. This can be done via the Save All context menu item. Also, the panel's context menu has two more items, Expand All and Collapse All, that expand and collapse all of the nodes of the call tree.
- The Event View panel has several improvements:
  - By default, AQtime detects all the exceptions that occur in the profiled application and logs them in the Event View panel. By using the new *Exception filter* settings you can specify Win32 exceptions that will be ignored. See *Exceptions in the Event View Panel*.
  - By using the new **Generate Process Dump** item of the Event View panel's toolbar, you can command AQtime to generate files that contain information about the profiled application, for example, a memory dump, CPU register values, information about loaded modules, threads and a call stack for each thread. You can then analyze the generated file in AQtrace or Visual Studio and explore what happens in your application. See *Generating Dumps for Profiled Applications*.
  - By using the new *Generate dump on exception* setting you can command AQtime to generate dumps on exceptions automatically.

- By using a single function call the profiled applications can now command AQtime to generate profiling results. See *Getting Results During Testing*.
- By using a new **Save Results to Source File** item of the Editor's context menu, you can store profiling results to source files directly from AQtime.
- The context menu of the Report panel has been extended. Now you can add the selected routines and classes not only to profiling areas but also to triggers and actions. See *Adding Selected Routines and Classes to Profiling Areas, Triggers and Actions.*
- For applications running under the .NET Framework ver. 2.0 or later, AQtime is now able to obtain user-defined names of CLR threads and display them in the Explorer panel.
- The Search Directories and Project Search Directories dialogs can now accept environment variables.
- The line profiling results shown in the Editor's gutter can be printed along with the source code. These results are transformed to comments that precede the source code lines.
- When comparing the results of several result sets, you can now use the Editor and Debugger panels that display the latest version of the application's source code. Thus, you can review the code of the routines whose performance was different during each profiling session.
- Interfaces that provide access to the AQtime engine via COM have been extended. Now the methods of the `IntegrationManager` object add a new `aqTimeIntegrationRunMode` object that you can select in the profiling mode, read/set run parameters and manage profiled modules. See *Working With AQtime via COM* for more information.

Technical Support and Resources

If you have questions, problems or just need help with AQtime, contact our **support team**. To submit your question, please use the Contact Support Form at:

http://www.automatedqa.com/support/message/

The support team will answer you via e-mail and all further communication will be made via e-mail. However, to start the conversation, please use the Contact Support Form.

For information on our support policies, please visit our web site www.automatedqa.com/support.

When sending a bug report, please specify the profiler you use and attach the Event View log. This information will help the support team find a solution to your problem more efficiently. To create the log:

- Switch on the *Exceptions* | *Active* option of the Event View panel (To set the option, right-click somewhere within the panel and select Options from the context menu).
- Start profiling and perform the actions that generated the problem.
- To save the log, choose **Select All** from the Event View context menu and then select either **Copy to Clipboard** or **Save to File**.

You may find answers to your question in the list of the **frequently asked questions** which is available at:

http://www.automatedqa.com/products/aqtime/faqs/

You can also ask questions, search for answers, exchange comments and suggestions on our **forums**:

- **http://www.automatedqa.com/forums**

Learn more about using AQtime from **technical papers** and **blogs** published at:

- **http://www.automatedqa.com/techpapers**
- **http://www.automatedqa.com/blogs**

Make sure you regularly visit the AutomatedQA Web site, **http://www.automatedqa.com/**, where you will find:

- News
- More recent support options including frequently asked questions on our products
- Technical papers from AutomatedQA
- Downloads, such as plug-ins and free tools, from AutomatedQA
- Hot Stuff contributed by experienced users and the AQA team (hands-on solutions, code, plug-ins, etc.)

Getting Started

Throughout the AQtime Help system, we will use the generic term *profiling* describing the use of any of AQtime profilers. Usually, but not always, a complete profiling operation involves the following steps:

- Compiling your application with debug information
- Opening your application in AQtime
- Controlling what to profile and when to profile
- Selecting the profiler to run
- Running the selected profiler and analyzing the results

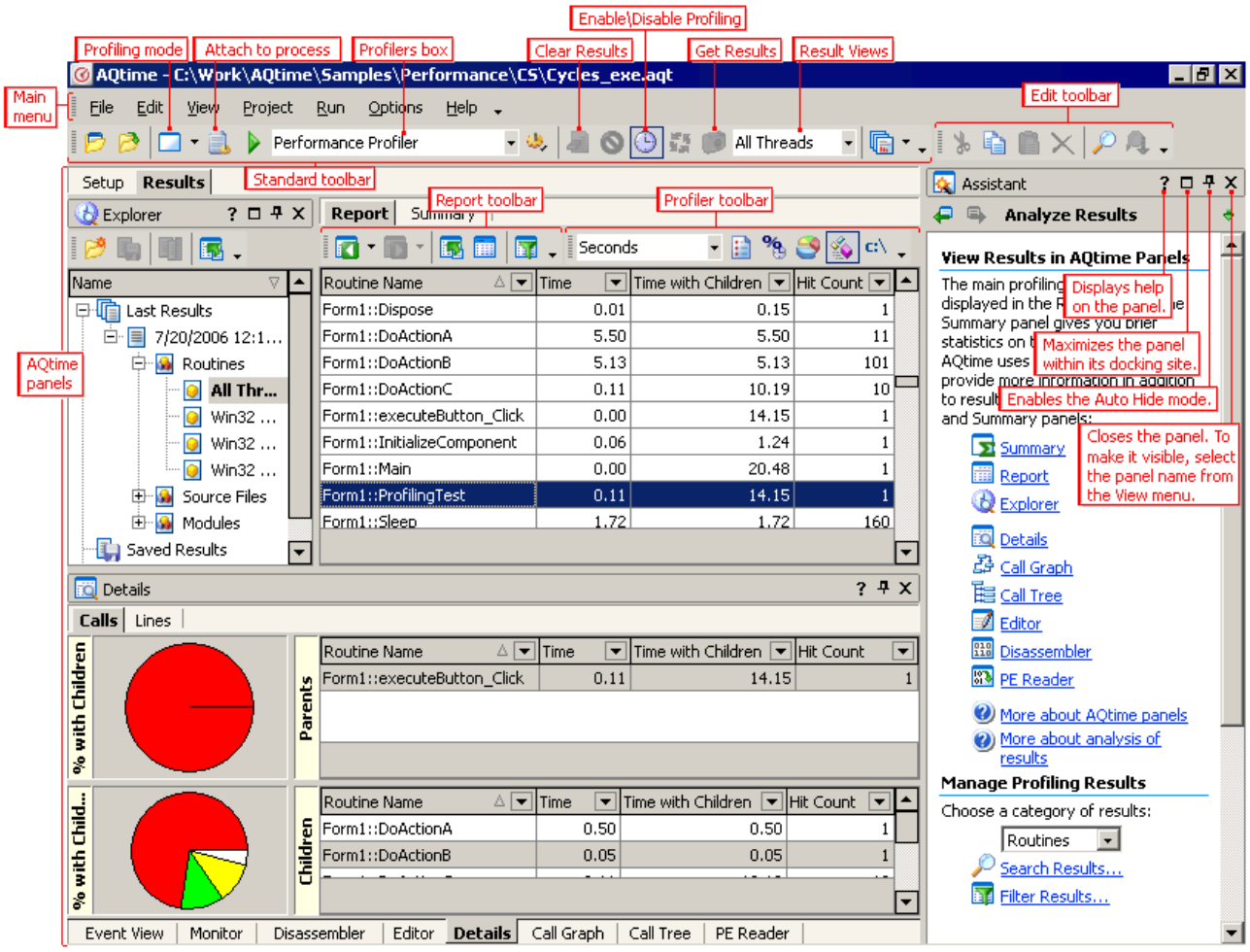
For more information on each step, read the Getting Started topics. They describe all main AQtime features you may need to know about to perform profiling as your needs dictate. Note, however, that the Getting Started topics describe general profiling approach. You may need to perform some additional operations depending on your application type. For instance, if you profile an ASP.NET application, you may need to select the appropriate profiling mode.

User Interface Overview

AQtime Standalone

AQtime's user interface consists of panels, the main menu and toolbars. The general layout is as follows:

© 2010 AutomatedQA Corp.



Most of AQtime's screen area is occupied by panels. Each panel serves a separate purpose in your work with AQtime. The purpose of each and how they work together is explained in a separate topic, which you should read: *AQtime Panels*.

The size and layout of panels are not fixed. You can change panel sizes by dragging the separator between them. But the most important point about handling panels is how they can be moved around - *docking*. Panels are where you do your actual work and get your results in AQtime. Docking is our way of providing you with the most flexible workspace for the particular task you are interested in. It means that the entire work area can be reconfigured at will, even beyond what is possible with toolbars (moving, hiding, etc.). Docking of panels in AQtime is similar to docking windows in Microsoft Visual Studio. For complete description, see *Docking* in on-line help.

There are common ways of arranging columns and lines in the grids, which most panels display. In addition, almost each panel has a number of options that you can modify in the **Options** dialog. The general organization of each panel has its own set of options, which you can modify in the User Interface dialog.

To save the current panel layout to a file, select **View | Desktop | Save Docking to File** from AQtime's main menu (by default, these files have the .qtdock extension). To load the panel layout from a file, select **View | Desktop | Load Docking from File**. To restore the default panel layout, select **View | Desktop | Restore Default Docking**. The **Save Desktop As** and **Load Desktop** items of the **View | Desktop** submenu will save and load the panel layout along with the toolbar settings.

The AQtime interface receives commands in four ways:

- through menus
- through popup menus (right-click, context-dependent)
- through toolbars
- through keyboard shortcuts

Keyboard shortcuts can be customized via the Customize Keyboard dialog. You can define your own shortcuts or select one of the predefined key mapping schemes: *MS Visual Studio IDE* or *Borland IDE*.

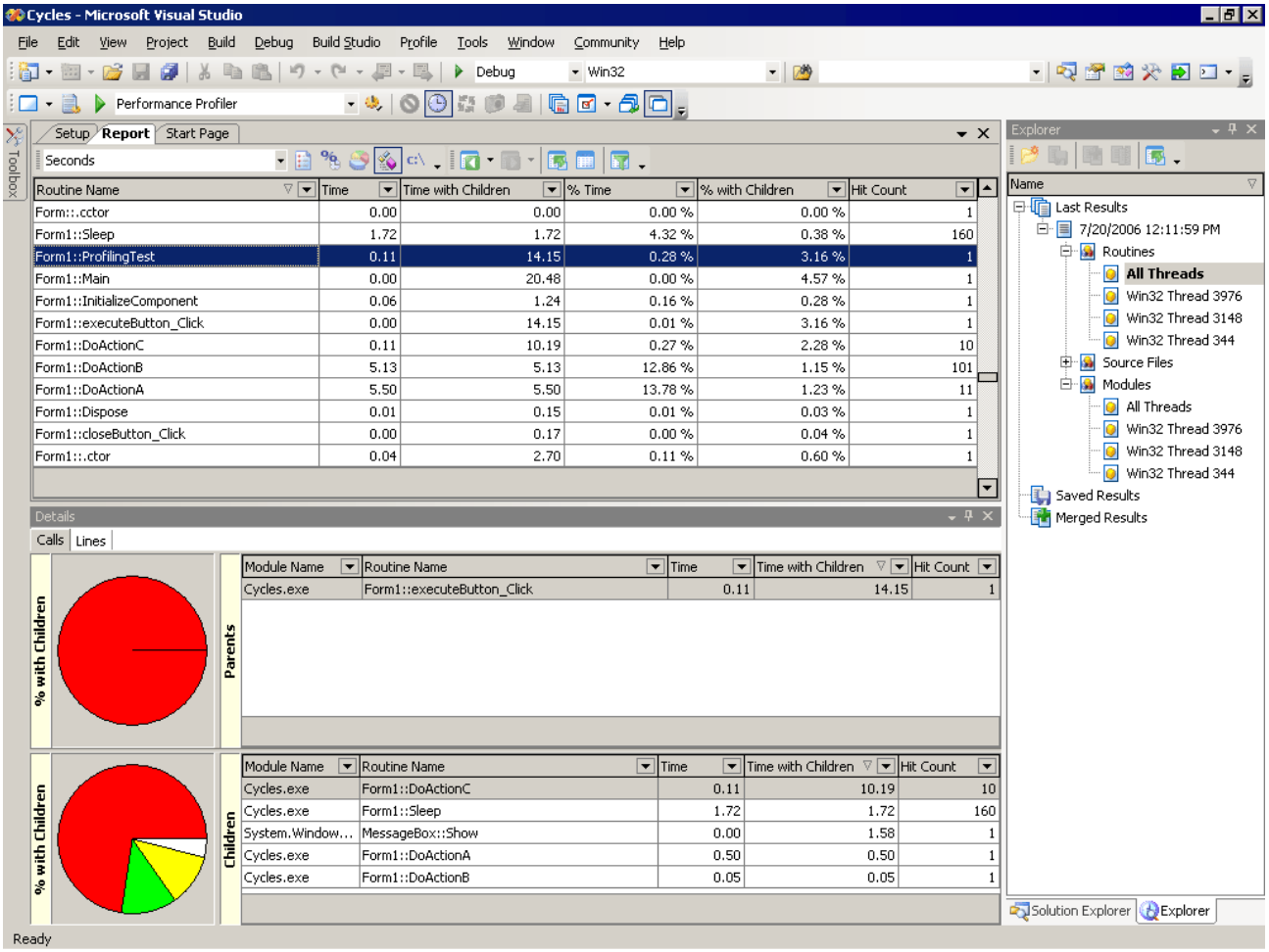
As in Microsoft Word or Excel, menus are a type of toolbar, and both can be customized at will. You can also create your own toolbars. By default, the Standard toolbar is docked to the top edge of the AQtime window. Other toolbars are docked to panels with which that toolbar works. For instance, the Setup toolbar is docked to the top edge of the Setup panel; the Report toolbar is docked to the top edge of the Report panel, etc. You can easily dock toolbar to any other edge by dragging them to the left, right or bottom edge of the panel. You can also dock the toolbars to any edge of the main window. See *Toolbars Customization* in on-line help for more information.

To remove or add buttons from toolbars and menus, you can either call the Toolbar Customization window or use the Quick Customization feature. For complete overview, see *Toolbars Customization* in on-line help.

To save or load the current layout of toolbars and toolbar items, use the **View | Desktop | Save Toolbars to File** and **View | Desktop | Load Toolbars from File** menu items. To restore the default toolbar layout, select **View | Desktop | Restore Default Toolbars**. To save and load the layout of panels, menus and toolbars, use the **View | Desktop | Save Desktop As** and **View | Desktop | Load Desktop** menu items.

AQtime Integrated into Visual Studio

AQtime's user interface consists of panels, the main menu and toolbars. Once AQtime has been integrated into Visual Studio .NET 2002, Visual Studio .NET 2003 or Visual Studio 2005, AQtime panels are listed in the Solution Explorer under the AQtime project node:


Cycles - Microsoft Visual Studio
File Edit View Project Build Debug Build Studio Profile Tools Window Community Help
Debug
Win32
Performance Profiler
Setup Report Start Page
Seconds
Routine Name Time Time with Children % Time % with Children Hit Count
Form::.cctor 0.00 0.00 0.00 % 0.00 % 1
Form1::Sleep 1.72 1.72 4.32 % 0.38 % 160
Form1::ProfilingTest 0.11 14.15 0.28 % 3.16 % 1
Form1::Main 0.00 20.48 0.00 % 4.57 % 1
Form1::InitializeComponent 0.06 1.24 0.16 % 0.28 % 1
Form1::executeButton_Click 0.00 14.15 0.01 % 3.16 % 1
Form1::DoActionC 0.11 10.19 0.27 % 2.28 % 10
Form1::DoActionB 5.13 5.13 12.86 % 1.15 % 101
Form1::DoActionA 5.50 5.50 13.78 % 1.23 % 11
Form1::Dispose 0.01 0.15 0.01 % 0.03 % 1
Form1::closeButton_Click 0.00 0.17 0.00 % 0.04 % 1
Form1::.ctor 0.04 2.70 0.11 % 0.60 % 1
Details
Calls Lines
% with Children
Parents
Module Name Routine Name Time Time with Children Hit Count
Cycles.exe Form1::executeButton_Click 0.11 14.15 1
% with Children
Children
Module Name Routine Name Time Time with Children Hit Count
Cycles.exe Form1::DoActionC 0.11 10.19 10
Cycles.exe Form1::Sleep 1.72 1.72 160
System.Window... MessageBox::Show 0.00 1.58 1
Cycles.exe Form1::DoActionA 0.50 0.50 1
Cycles.exe Form1::DoActionB 0.05 0.05 1
Explorer
Name
Last Results
7/20/2006 12:11:59 PM
Routines
All Threads
Win32 Thread 3976
Win32 Thread 3148
Win32 Thread 344
Source Files
Modules
All Threads
Win32 Thread 3976
Win32 Thread 3148
Win32 Thread 344
Saved Results
Merged Results
Solution Explorer
Explorer
Ready

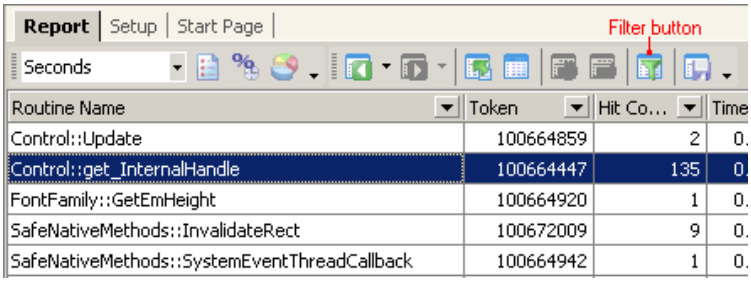
The panels are grouped by their role in your AQtime project. Panels are where you do your actual work and get your results in AQtime. Every panel serves a different purpose. For more detailed information on the different purposes and on how the panels work together, see *AQtime Panels*.

To bring up a panel, either select it in the Solution Explorer; or select **Profile | Options | Panel List** from Visual Studio's menu and then choose the panel from the ensuing Select Panel dialog (the Profile menu item is also added by AQtime. See below).

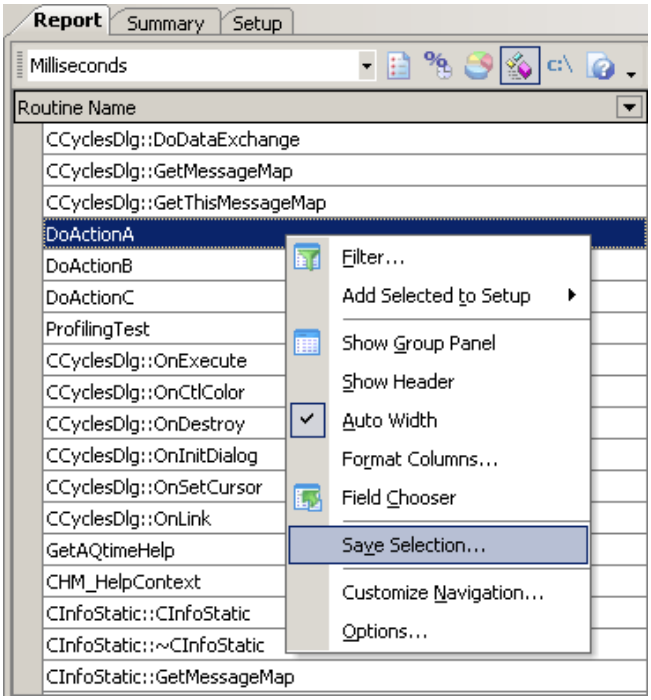
You can change the panel size and location in the same way you would with other Visual Studio windows. There are common ways of arranging columns and lines in the grids, which most panels display. In addition, almost each panel has a number of options you can modify in the **Options** dialog.

The **Profile | Toggle Panels** menu item lets you quickly hide or display AQtime panels. If there are visible AQtime panels, then pressing this item will hide them. If there are no visible AQtime panels, pressing this item will show the panels that were visible at the moment of hiding.

Most panels have a toolbar at the top. The toolbar items allow you to perform certain operations over data displayed in panels. For instance, the **Filter** item of the Report toolbar displays the Filter dialog where you can create a filter condition and apply it to profiler results:

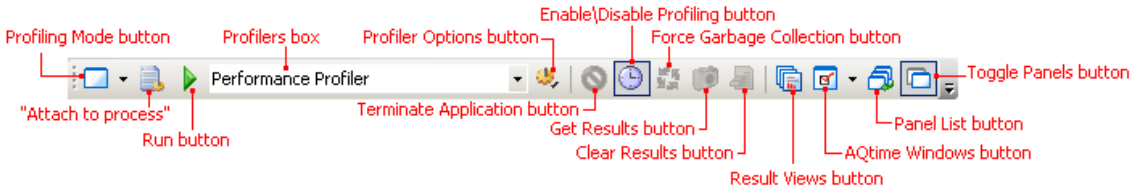


Even more operations are available in the context menu for each panel. For instance, the context menu of the Report panel holds the **Save Selection** item that exports profiling results to text, Excel, html or xml files:



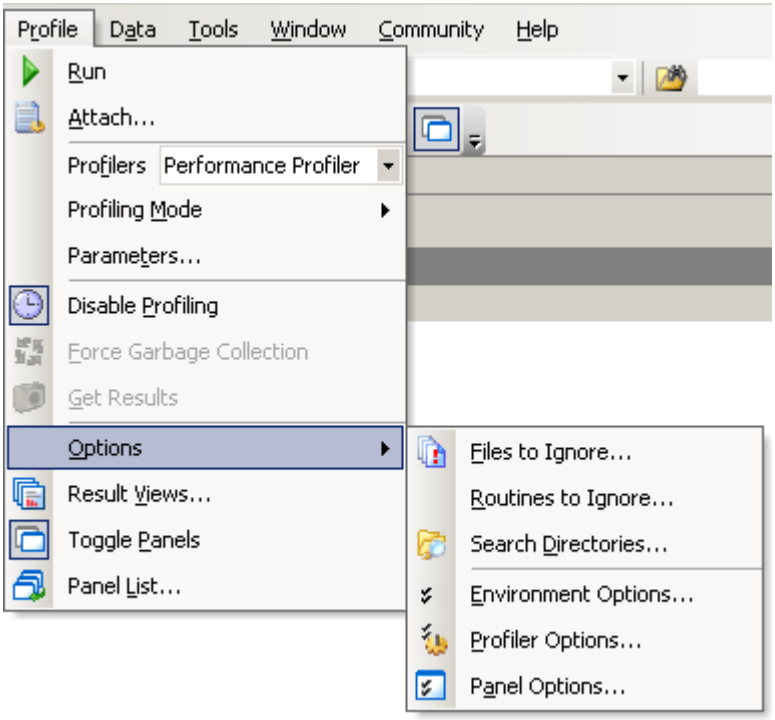
You can dock a "panel" toolbar to any side of the panel that holds this toolbar. There is also a way to hide or display items of these toolbars. For more information, see *Toolbar Customization* in on-line help. Unlike toolbars, the panels' context menus are not customizable.

Besides the "panel" toolbars, AQtime also adds one more toolbar, **AQtime Standard**, to Visual Studio. You can display this toolbar by right-clicking somewhere in the toolbar area and checking **AQtime** from the subsequent context menu. The toolbar holds the following items:

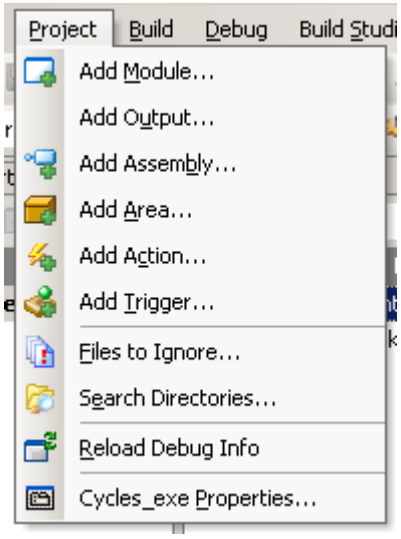


The toolbar items are displayed or hidden depending on the current context. For instance, the **Terminate** item will not be visible until you start profiling.

AQtime also adds the **Profile** menu item to Visual Studio's main menu. This menu holds the same items as the AQtime Standard toolbar. The only "extra" item is Options: it opens the context menu that provides access to AQtime dialogs:



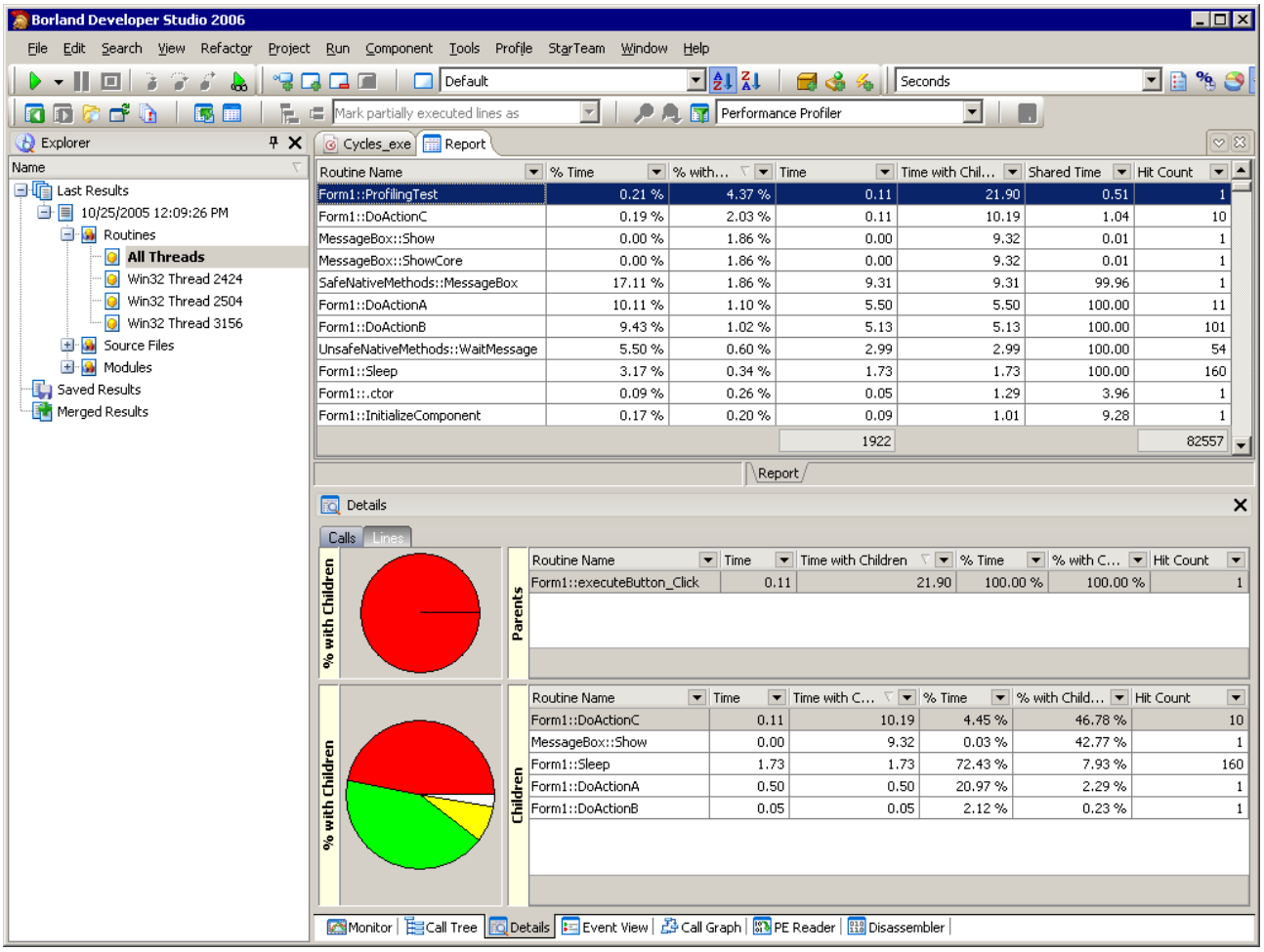
In addition to the Profile menu, AQtime inserts several items to the **Project** menu:



You can manage the Profile and Project menus and the AQtime Standard toolbar in the same manner as you manage other Visual Studio menus and toolbars.

AQtime Integrated into Borland Developer Studio

AQtime's user interface consists of panels, the main menu and toolbars. Once AQtime has been integrated into Borland Developer Studio 2006, all these elements of AQtime's user interface are displayed within the Borland Developer Studio environment.


Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Default
Seconds
Mark partially executed lines as
Performance Profiler
Explorer
Name
Last Results
10/25/2005 12:09:26 PM
Routines
All Threads
Win32 Thread 2424
Win32 Thread 2504
Win32 Thread 3156
Source Files
Modules
Saved Results
Merged Results
Cycles_exe
Report
Routine Name % Time % with... Time Time with Chil... Shared Time Hit Count
Form1::ProfilingTest 0.21 % 4.37 % 0.11 21.90 0.51 1
Form1::DoActionC 0.19 % 2.03 % 0.11 10.19 1.04 10
MessageBox::Show 0.00 % 1.86 % 0.00 9.32 0.01 1
MessageBox::ShowCore 0.00 % 1.86 % 0.00 9.32 0.01 1
SafeNativeMethods::MessageBox 17.11 % 1.86 % 9.31 9.31 99.96 1
Form1::DoActionA 10.11 % 1.10 % 5.50 5.50 100.00 11
Form1::DoActionB 9.43 % 1.02 % 5.13 5.13 100.00 101
UnsafeNativeMethods::WaitMessage 5.50 % 0.60 % 2.99 2.99 100.00 54
Form1::Sleep 3.17 % 0.34 % 1.73 1.73 100.00 160
Form1::.ctor 0.09 % 0.26 % 0.05 1.29 3.96 1
Form1::InitializeComponent 0.17 % 0.20 % 0.09 1.01 9.28 1
1922 82557
Report
Details
Calls Lines
% with Children
Parents
Routine Name Time Time with Children % Time % with C... Hit Count
Form1::executeButton_Click 0.11 21.90 100.00 % 100.00 % 1
% with Children
Children
Routine Name Time Time with C... % Time % with Child... Hit Count
Form1::DoActionC 0.11 10.19 4.45 % 46.78 % 10
MessageBox::Show 0.00 9.32 0.03 % 42.77 % 1
Form1::Sleep 1.73 1.73 72.43 % 7.93 % 160
Form1::DoActionA 0.50 0.50 20.97 % 2.29 % 1
Form1::DoActionB 0.05 0.05 2.12 % 0.23 % 1
Monitor Call Tree Details Event View Call Graph PE Reader Disassembler

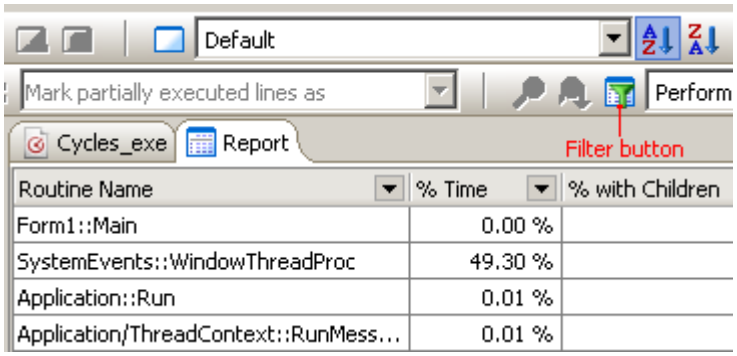
The panels are grouped by their role in your AQtime project. Panels are where you do your actual work and get your results in AQtime. Every panel serves a different purpose. For more detailed information on the different purposes and on how the panels work together, see *AQtime Panels*.

To bring up any of AQtime's panels, select it in the **Profile Windows** submenu of Borland Developer Studio's **View** menu.

You can change the panel size and location in the same way you would with other Borland Developer Studio windows. There are common ways of arranging columns and lines in the grids, which most panels display. In addition, almost each panel has a number of options you can modify in the **Options** dialog.

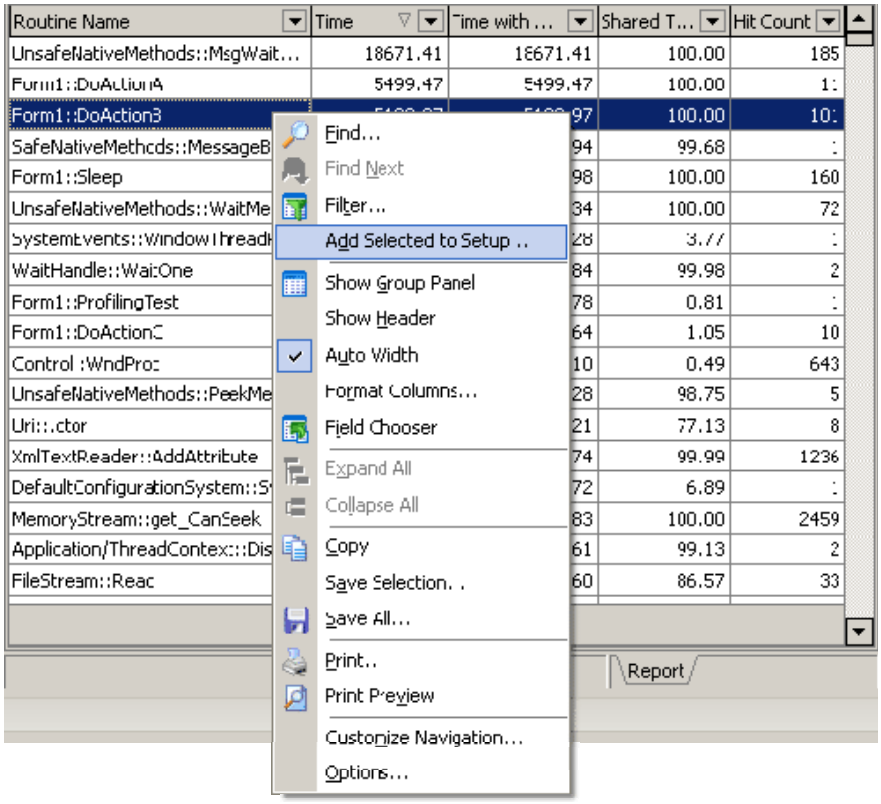
Most panels have a toolbar associated with them. These toolbars are displayed at the top of the Borland Developer Studio window. The toolbar items allow you to perform certain operations with data displayed in panels. For instance, the **Filter** item of the **Report.AQtime** toolbar displays the **Filter** dialog where you can create a filter condition and apply it to profiler results:

© 2010 AutomatedQA Corp.



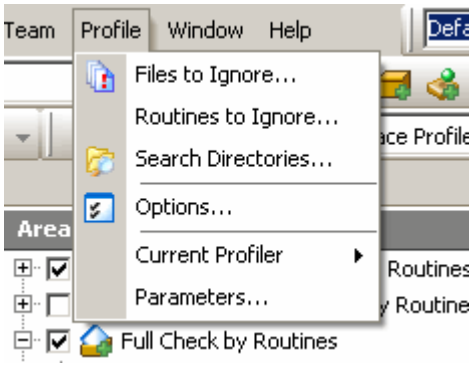
The toolbar items are displayed or hidden depending on the current context. For instance, the **Terminate** item will not be visible until you start profiling.

Even more operations are available in the context menu for each panel. For instance, the context menu of the **Report** panel holds items (**Save Selection** and **Save All**) that are useful for exporting profiler results to text, Excel, html or xml files:

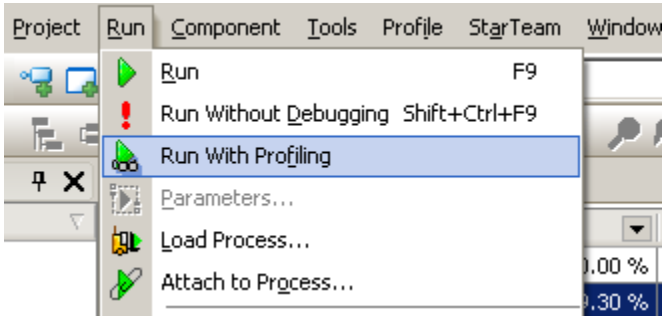


There is a way to hide or display items of these toolbars. For more information, see *Toolbar Customization* in on-line help. Note that in order to distinguish AQtime's toolbars from the others in Borland Developer Studio's **Customize** dialog, AQtime's toolbars have the **.AQtime** postfix in their names (for instance, **Run.AQtime**). Unlike toolbars, the panels' context menus are not customizable.

AQtime also adds the **Profile** menu item to Borland Developer Studio's main menu. This menu holds the items that open dialogs used to configure AQtime's general options which are not specific to a particular AQtime project:



In addition to the Profile menu, AQtime inserts the **Run With Profiling** item to the **Run** menu:



Selecting this menu item or pressing the corresponding **Run With Profiling** button starts profiling within the default profiling areas: *Profile Entire .NET Code* and *Full Check*.

The **Run With Profiling** command is available even when an AQtime project is not added to the current project group, in this case AQtime creates a new project and starts profiling immediately.

Note: The described behavior has a side effect; if a project has custom profiling areas, they are disabled and only *Profile Entire .NET Code* and *Full Check* areas become enabled. When you want to use custom areas, rather than using the **Run With Profiling** command, start the profiling as described below:

- Select **Run** menu item or click **Run** button on the **Debug** toolbar or press F9, while the AQtime project is active in Borland Developer Studio's **Project Manager**.
- Right-click the AQtime project in the **Project Manager** and select **Start** from the context menu.

AQtime Panels

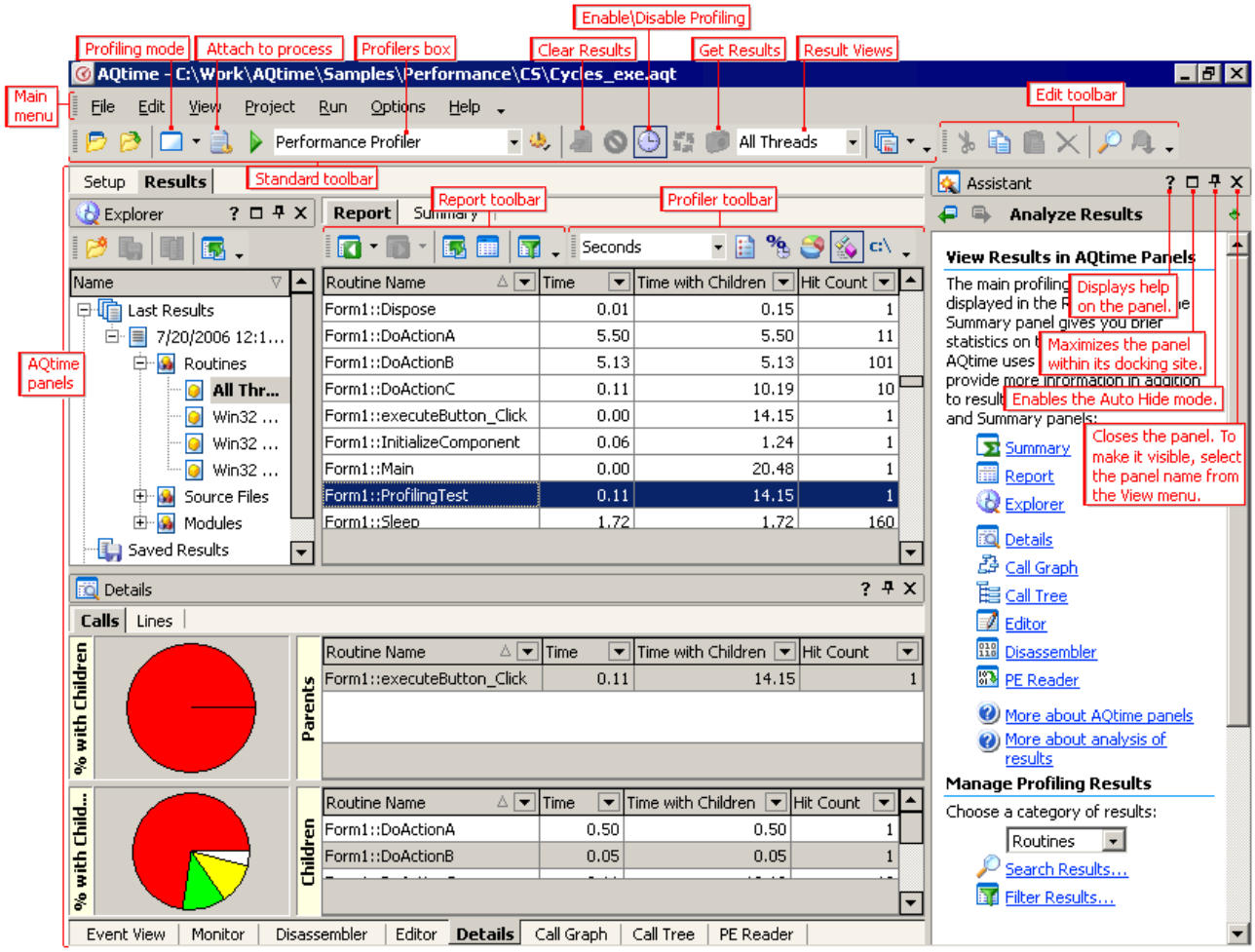
When using AQtime --

- First you define a profiling project, which will likely involve many profile runs over several days or months.
- Then, for each profile run --
  - you first define what you wish to profile, …
  - then execute the profile run, …

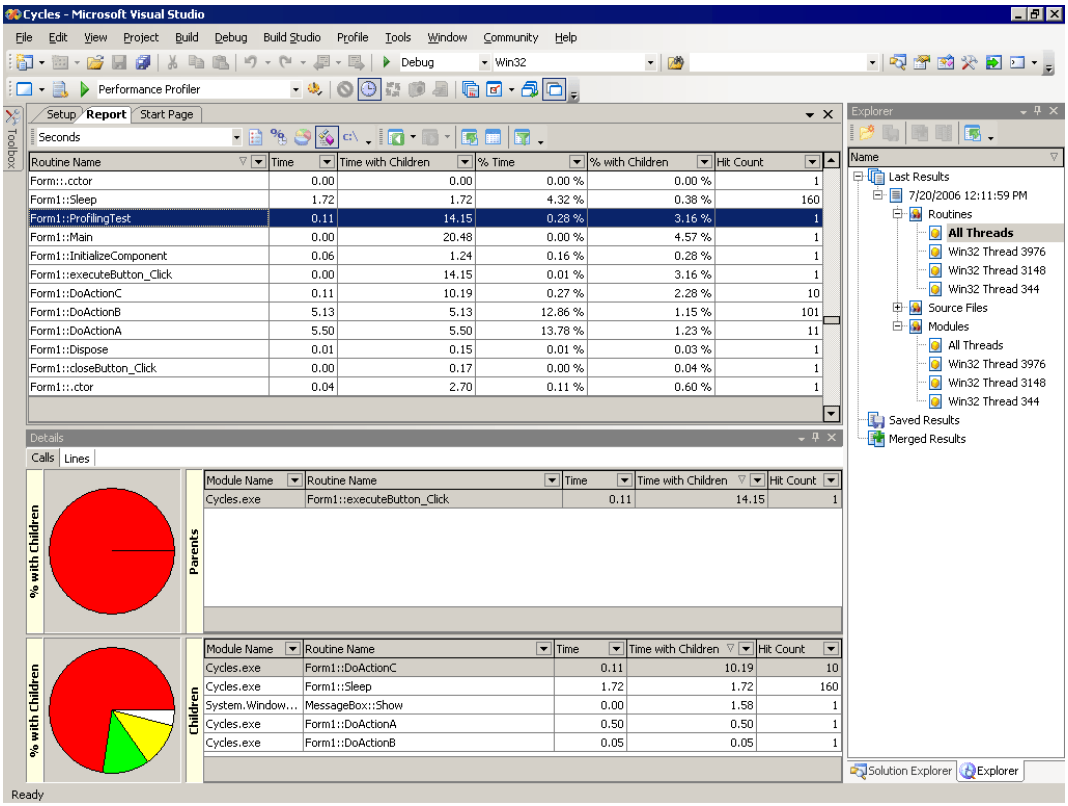
  - which generates results when the application exits or when you ask for this through **Run | Get Results** (**Profile | Get Results** in Visual Studio or **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar).
- Once you have the new results --
  - you can browse through them …
  - or examine them in specific, targeted ways.
- This result set is automatically added to the collected result sets for the project, and then or later --
  - you can manage the collection, …
  - examine stored results with all the tools available for new results, …
  - and compare the result sets.

You will spend most of your time in AQtime working in its panels. The panels are organized to support the task list above. Of all the tasks above, only the first, defining a project is done outside a panel.

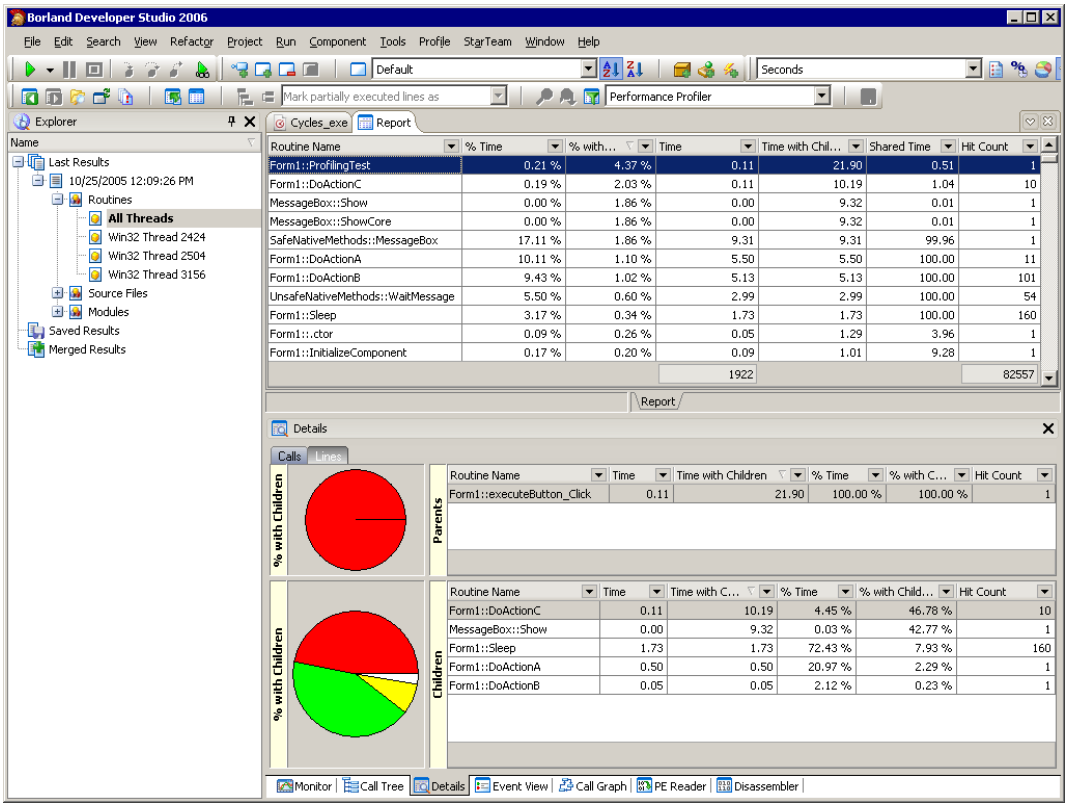
In the following picture, the latest result set from the Explorer panel is being browsed through in the Report panel. Extensive details, for the line currently selected in the Report panel, are displayed below in the Details panel.



Panels in AQtime standalone


Cycles - Microsoft Visual Studio
File Edit View Project Build Debug Build Studio Profile Tools Window Community Help
Debug
Win32
Performance Profiler
Setup Report Start Page
Seconds
Routine Name Time Time with Children % Time % with Children Hit Count
Form::.cctor 0.00 0.00 0.00 % 0.00 % 1
Form1::Sleep 1.72 1.72 4.32 % 0.38 % 160
Form1::ProfilingTest 0.11 14.15 0.28 % 3.16 % 1
Form1::Main 0.00 20.48 0.00 % 4.57 % 1
Form1::InitializeComponent 0.06 1.24 0.16 % 0.28 % 1
Form1::executeButton_Click 0.00 14.15 0.01 % 3.16 % 1
Form1::DoActionC 0.11 10.19 0.27 % 2.28 % 10
Form1::DoActionB 5.13 5.13 12.86 % 1.15 % 101
Form1::DoActionA 5.50 5.50 13.78 % 1.23 % 11
Form1::Dispose 0.01 0.15 0.01 % 0.03 % 1
Form1::closeButton_Click 0.00 0.17 0.00 % 0.04 % 1
Form1::.ctor 0.04 2.70 0.11 % 0.60 % 1
Details
Calls Lines
% with Children
Parents
Module Name Routine Name Time Time with Children Hit Count
Cycles.exe Form1::executeButton_Click 0.11 14.15 1
% with Children
Children
Module Name Routine Name Time Time with Children Hit Count
Cycles.exe Form1::DoActionC 0.11 10.19 10
Cycles.exe Form1::Sleep 1.72 1.72 160
System.Window... MessageBox::Show 0.00 1.58 1
Cycles.exe Form1::DoActionA 0.50 0.50 1
Cycles.exe Form1::DoActionB 0.05 0.05 1
Explorer
Name
Last Results
7/20/2006 12:11:59 PM
Routines
All Threads
Win32 Thread 3976
Win32 Thread 3148
Win32 Thread 344
Source Files
Modules
All Threads
Win32 Thread 3976
Win32 Thread 3148
Win32 Thread 344
Saved Results
Merged Results
Solution Explorer Explorer
Ready

AQtime panels integrated into Visual Studio


Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Default
Seconds
Mark partially executed lines as
Performance Profiler
Explorer
Name
Last Results
10/25/2005 12:09:26 PM
Routines
All Threads
Win32 Thread 2424
Win32 Thread 2504
Win32 Thread 3156
Source Files
Modules
Saved Results
Merged Results
Cycles_exe Report
Routine Name % Time % with... Time Time with Chil... Shared Time Hit Count
Form1::ProfilingTest 0.21 % 4.37 % 0.11 21.90 0.51 1
Form1::DoActionC 0.19 % 2.03 % 0.11 10.19 1.04 10
MessageBox::Show 0.00 % 1.86 % 0.00 9.32 0.01 1
MessageBox::ShowCore 0.00 % 1.86 % 0.00 9.32 0.01 1
SafeNativeMethods::MessageBox 17.11 % 1.86 % 9.31 9.31 99.96 1
Form1::DoActionA 10.11 % 1.10 % 5.50 5.50 100.00 11
Form1::DoActionB 9.43 % 1.02 % 5.13 5.13 100.00 101
UnsafeNativeMethods::WaitMessage 5.50 % 0.60 % 2.99 2.99 100.00 54
Form1::Sleep 3.17 % 0.34 % 1.73 1.73 100.00 160
Form1::.ctor 0.09 % 0.26 % 0.05 1.29 3.96 1
Form1::InitializeComponent 0.17 % 0.20 % 0.09 1.01 9.28 1
1922 82557
Report
Details
Calls Lines
% with Children
Parents
Routine Name Time Time with Children % Time % with C... Hit Count
Form1::executeButton_Click 0.11 21.90 100.00 % 100.00 % 1
% with Children
Children
Routine Name Time Time with C... % Time % with Child... Hit Count
Form1::DoActionC 0.11 10.19 4.45 % 46.78 % 10
MessageBox::Show 0.00 9.32 0.03 % 42.77 % 1
Form1::Sleep 1.73 1.73 72.43 % 7.93 % 160
Form1::DoActionA 0.50 0.50 20.97 % 2.29 % 1
Form1::DoActionB 0.05 0.05 2.12 % 0.23 % 1
Monitor Call Tree Details Event View Call Graph PE Reader Disassembler

AQtime panels integrated into Borland Developer Studio

There are four major panels. They closely follow the task list above:

Panel	Description
Setup	This is where you go before a profile run to define what it will profile and when, once you have selected which profiler to use from the **Profiler** dropdown list.
Event View	This reports messages and events during profiling as they occur. In other words, this is where you track the ongoing profile run.
Report	After your results are generated, they are displayed here, and you can browse through them. If the profiled application used threads, the **Thread** dropdown list will allow you choose any single thread to display the results for, or all threads. There are also ways to filter the results and to organize the display in the panel. You can save a particular format for the panel and the filters as a result view.
Explorer	This is where you manage result sets from the current project, including the latest. Normally, the sets displayed are only those for the currently selected profiler, but you can also choose to have all the collections (one per profiler) presented in a tree view. Any result set can be selected and displayed in the Report panel. You can add a description to each set, and you can store it, retrieve it, compare it to others, save it to a separate file or read it from one. You can organize the entire collection through folders, and you can delete sets from it. Results in each single result set are organized into several categories in the Explorer panel. For instance, on the picture above, results of the Performance profiler are shown per thread. The categories depend on the profiler in use, for example, categories used by the Performance profiler differ from the Allocation profiler categories.

Six more panels act as extensions to the Report panel, providing various types of information about the line currently selected in the Report panel, or about global results:

Panel	Description
Details	AQtime profilers use this panel to provide additional information for a selected row in the Report panel, which would be impossible to show within reasonable space as additional columns in the Report panel itself.
Call Graph	This panel shows the callers for the routine selected in Report, and which routines it called in turn, with statistics for each link. The call hierarchy can be browsed up or down by double-clicking on any parent or child, without returning to Report.
Call Tree	This panel includes two tabbed pages showing execution *paths* for the routine selected in the Report panel. One of the pages, **Parents**, displays all stacks of function calls that led to the call to the selected routine. Another page, **Children**, displays all function calls that were initiated by the selected routine. Both panels highlight the “longest” path (for example, the path that took most time to execute) to help you find bottlenecks faster.
Editor	Displays the source code for the line selected in Report (if available), along with optional summary results. This panel is only available if AQtime is running as a standalone application. If AQtime is running as a package within Microsoft Visual Studio, Visual Studio’s native **Code Editor** is used instead of AQtime’s Editor. If AQtime is running as a package within Borland Developer Studio, Borland Developer Studio’s native **Editor** is used instead of AQtime’s Editor.
Summary	This panel holds a summary of the profiling results. The contents of the panel depend

	on the current profiler. Use it to quickly find routines and classes that need to be optimized.
Disassembler	Displays the binary code for the routine that is selected in the Report, Details, Event View, Call Graph, Call Tree, Setup or Summary panels, in assembly language, showing either the source code with its line-by-line disassembly, or plain disassembly from the binary code in memory.

AQtime includes three more panels: Assistant, PE Reader and Monitor.

Assistant	This panel helps you get started using AQtime quickly. Depending on which step you are currently at in your project, it displays information that helps you use all the power that AQtime's features can provide at this step. This panel can even be helpful to AQtime-gurus, since it provides faster access to AQtime features.
PE Reader	The PE Reader panel provides information about executables used by the main module of your AQtime project. It lets you easily see which modules are linked to your application at load-time and thus determine the modules that are necessary for your application to function properly. PE Reader provides information about module versions, imported and exported routines, etc.
Monitor	The Monitor panel is used along with the Allocation profiler. It traces memory allocations in real time and displays the size of allocated memory blocks, the number of existing object instances, etc. during the application run. It is very easy to use and quite instructive at times.

AQtime keeps information about your browsing in its panels, just as a Web browser would. There are the **Display Previous** and **Display Next** buttons on the Report toolbar, and you can use them to move back and forth among a sequence of routines that you are focusing on.

Most AQtime panels hold tables of data. You can customize them as you wish: change the column size and place, add and remove columns, sort and group records, etc. See *Arranging Columns, Lines and Panels* in on-line help. Exactly what each panel displays is configurable through **Options | Panel Options** from the main menu. There are separate options for each panel.

If you run AQtime as a standalone application, you can undock each panel and move it to any other location. The **View | Desktop | Docking Allowed** menu item specifies if the docking is active or not. If this option is on, you can undock any panel by dragging its header. You can then drag this panel to another location, for example, you can put it on a tabbed page along with another panel. See *Docking* in on-line help for complete description of the docking mechanism. If you ever need to bring up a panel quickly, the **View** menu was made specifically for that reason - it's your failsafe panel retriever.

If you use AQtime integrated into Visual Studio, you will see that AQtime panels are fully integrated into Visual Studio's IDE. You can dock, undock and move them around just as you would any other Visual Studio window. To bring up a panel quickly, simply select it in the Solution Explorer. You can also bring up a panel by selecting **Profile | Panel List** from Visual Studio's menu and choosing the panel name in the ensuing dialog.

If you use AQtime integrated into Borland Developer Studio, you will see that AQtime panels are fully integrated into Borland Developer Studio's IDE. To bring up a panel quickly, simply select it in the **View | Profile Windows** menu.

Note that when you start profiling in Visual Studio, AQtime hides panels visible at design time and shows panels visible at profile time. After the profiling is over, AQtime hides the profile-time panels and displays design-time panels. Both profile-time and design-time collections of panels can be changed at your desire. If a panel is visible when profiling starts, it is automatically added to the design-time panel collection. If you make a panel visible at profile time and this panel is visible when profiling completes, it will be automatically added

to the profile-time panel collection. The **Profile | Toggle Panels** menu item in Visual Studio lets you quickly hide or display AQtime panels. If there are visible AQtime panels, then pressing this item will hide them. If there are no visible AQtime panels, pressing this item will show the panels that were visible at the moment of hiding.

Most panels of AQtime have toolbars associated with them. The toolbar items are used to perform certain operations on data that are displayed in the panel. For instance, the **Field Chooser** item on the Report toolbar displays a list of available columns allowing you to add a column to the panel by dragging it from this list to the panel. You can dock a toolbar to any side of the panel, to which this toolbar belongs. For more information, see *Toolbars Customization* in on-line help.

AQtime Profilers

This topic is actually an extended section of the *AQtime Overview*, so it is we recommend you read that first. It aims to supply a brief answer to the question: What do you use the profilers for?

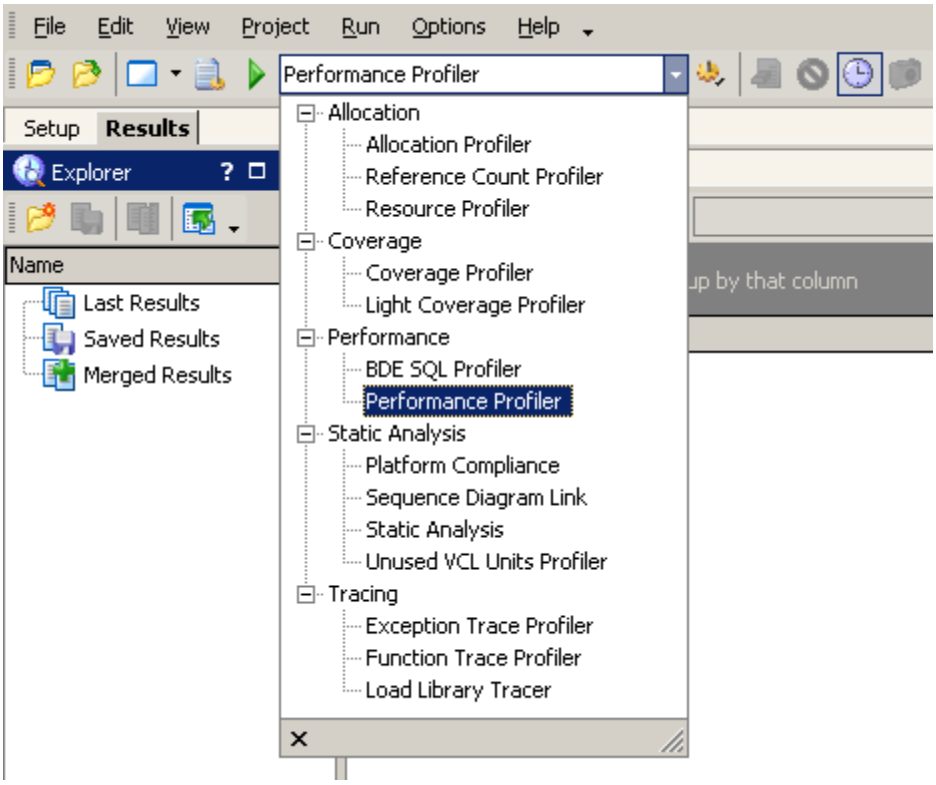
AQtime includes fourteen profilers: **Performance**, **BDE SQL, Reference Count, Allocation**, **Resource**, **Coverage**, **Light Coverage**, **Exception Trace**, **Function Trace**, **Load Library Tracer**, **Static Analysis**, **Sequence Diagram Link**, **Unused VCL Units** and **Platform Compliance**. Four profilers - Static Analysis, Sequence Diagram Link, Platform Compliance and Unused VCL Units - do not run your application. Static Analysis, Sequence Diagram Link and Unused VCL Units explore debug information and Platform Compliance analyzes the import function table included in the executable. The other profilers launch your application. Performance, BDE SQL, Reference Count, Allocation, Resource, Coverage, Load Library Tracer and Function Trace gather data while the application runs and provide results when the run is over or when you select **Run | Get Results** from AQtime's menu (**Profile | Get Results** from Visual Studio's menu or **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar). Unlike these profilers, Exception Trace displays results in real time.

All the profilers can profile both managed (.NET) and unmanaged (native) modules. The only exception is Platform Compliance - it can profile only unmanaged code. All profilers support 64-bit application profiling. Also, AQtime is capable of profiling COM, ASP.NET, IIS and service applications under x64 platform.

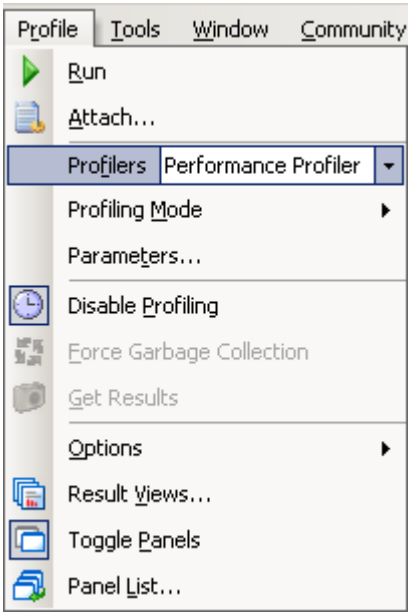
If your application is a .NET application, then AQtime profilers do not need any more than normal compilation, unless you wish to profile routines at line level or unless you wish to have direct access to source code for methods or classes listed in profiler results (see *How the AQtime Profilers Use Metadata and Debug Information*).

If your application is a native-code (that is, non-.NET) application, it must be compiled with debug information. For more information, see *How the AQtime Profilers Use Metadata and Debug Information*.

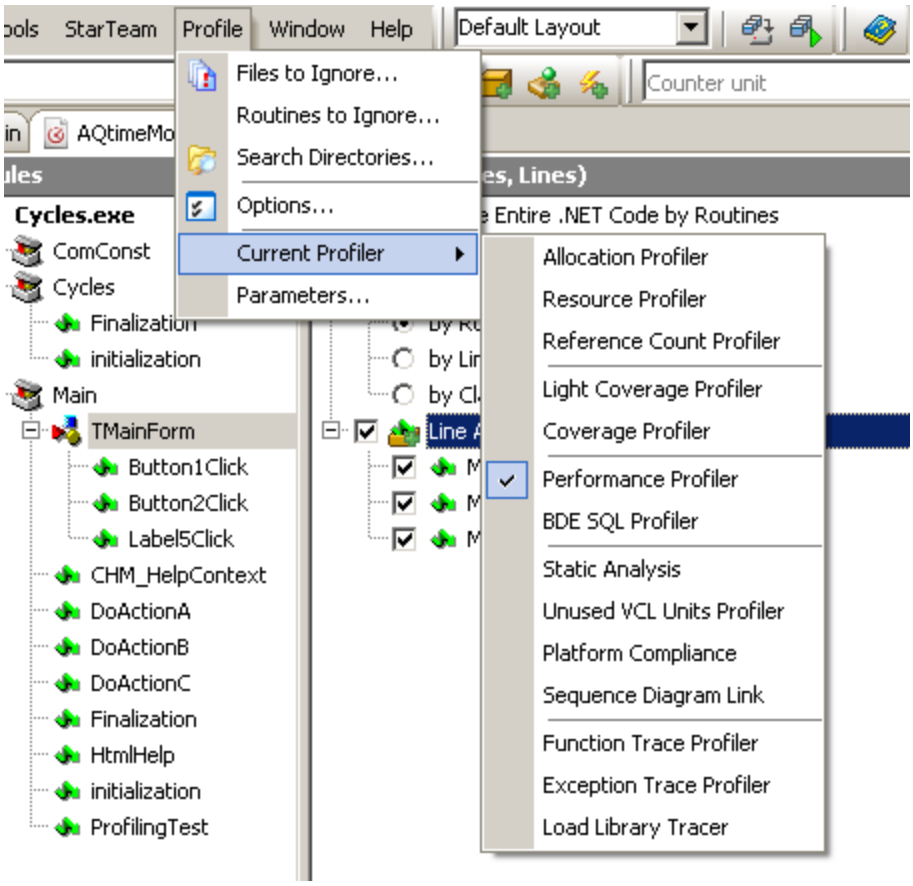
You select the profiler to be used for application analysis from the Standard toolbar (or from the **Profiler** dropdown list in the **Profile** menu, if you use Visual Studio, or from the **Current Profiler** submenu of the **Profile** menu, if you use Borland Developer Studio):



AQtime Standalone



AQtime integrated into Microsoft Visual Studio



AQtime integrated intoBorland Developer Studio

Here is some brief information on profilers:

- The **Performance** profiler is meant to be used to find bottlenecks in your application and for determining what causes these bottlenecks. During the run, this profiler gathers lots of statistics on each routine included in profiling tasks: how many times the routine was called, what function called the routine and what functions it called, how many exception occurred during the routine execution, etc. In addition, the profiler also measures such application characteristics as the function execution time and the number of CPU cache updates. The value the profiler measures depends on the *Active counter* option. Currently, the profiler includes the following counters:
  - *Elapsed Time*
  - *User Time*
  - *User+Kernel Time*
  - *CPU Mispredicted Branches*
  - *CPU Cache Misses*
  - *Context Switches*
  - *64K Aliasing Conflicts*
  - *Split Load Replays*
  - *Split Store Replays*
  - *Blocked Store Forwards Replays*
  - *Soft Memory Page Faults*
  - *Hard Memory Page Faults*
  - *All Memory Page Faults*

  The *Elapsed Time*, *User Time* and *User+Kernel Time* counters time application functions. Depending on the counter in use, the resultant time may (or may not) include time spent on executing the operating system code, time spent on switching between threads, etc. The *CPU Mispredicted Branches* counter reports whether your code can be well predicted by the CPU's branch prediction unit. The *CPU Cache Misses* counter lets you determine whether hotspots in

your applications are caused by an excessive number of CPU cache updates. The *Context Switches* counter calculates the number of context switches that occur during the function execution. Other counters let you determine whether your application algorithms used to work with memory are effective and do not cause performance bottlenecks. For a complete description of the counters and counter limitations, see *Counters Overview*.

For .NET applications, the Performance profiler also lets you determine how much the .NET runtime contributes to function results: the profiler measures the time spent for Just-in-Time compilation and garbage collection and displays these times as <*JIT compiler*> and <*Garbage collector*> routines in the profiler results.

These routines as well as counters and certain columns in profiler results help you determine what caused a bottleneck during the application run. That is, you can use the Performance profiler not just to establish the fact that a bottleneck exists, but to find the cause of the bottleneck as well. For more information on this, see *Searching for Bottleneck Reasons With the Performance Profiler*.

The Performance profiler can analyze your code at two levels of detail: routine and line. To profile routines at line level, the application must be compiled with debug information. See *How AQtime Profilers Use Metadata and Debug Information*.

The profiler collects separate results for each thread in a multithreaded application. It can organize results by operating system threads as well as by .NET runtime threads. See *Profiling Multiple Threads* for more information.

We would like to note once again that Performance profiler supports the profiling of both managed and native (i.e. unmanaged) modules (see also *Profiling Managed and Unmanaged Code*). This lets you profile, for example, unmanaged dynamic link libraries along with the .NET modules that use these libraries.

- The **Allocation** profiler traces the memory usage within your applications during the profiler run. It reports how many objects of each class exist, how many memory blocks were allocated, how much memory that objects and blocks occupy, etc. The profiler gathers lots of information: it traces call stacks for objects and memory blocks, determines references between different managed objects, etc. Using the Allocation profiler you can easily find memory leaks in your unmanaged (i.e. non-.NET) applications. As for .NET applications, then though the common language runtime reclaims all memory allocated for objects when the application is over, this profiler is useful when trying to track the objects during the application run. Using the **Monitor** panel when running the Allocation profiler, you can view the allocation information in charts or histograms, which helps you trace the memory usage in real time.
- The **Resource** profiler follows how your application exploits Windows resources (fonts, brushes, bitmaps, and other graphic components, registry, COM objects, print spooler, etc.) during the profiler run. It reports what these resources are, how many resources of each type were created up to given moment, how many of them still exist, how much memory is occupied by the resources in use, what errors in resource management functions occurred during the run, etc. The profiler can help you find resource leaks (unreleased resources) and resource errors in managed and unmanaged applications. For each occupied resource instance, the profiler keeps its call stack of functions calls, which lets you easily discover how this resource instance was allocated.
- The **Reference Count** profiler tracks the number of references to COM objects that implement one or several interfaces. The profiler traces the creation and deletion of references and allows you to pinpoint unreleased references or those that were released prematurely.
- The **Coverage** profiler determines whether the function or line was executed during the application run. It also counts the number of times a routine (or line) was executed during the

profiler run. Using this profiler you can easily find what application areas your tests “cover” and what was left untested.

- The **Light Coverage** profiler determines whether a routine or a line was executed during the profiler run. This profiler is similar to the **Coverage** profiler but it does not track the hit count and it does not allocate results by threads.
- The **Static Analysis** profiler will tell you which methods exist in the application, where they are called from in the source code and what they call in turn. This does not tell you if and when the calls will execute, but it does give a full report of method inter-dependence in the source. See it as an intelligent overview browser of the debug information that is linked into the executable.

  A powerful addition to the Static Analysis profiler is the **PE Reader** panel. It also performs the analysis of your application statically and provides detailed information about modules used by the application. For example, it shows tables of imported and exported routines, module base addresses, entry points of routines and their offsets in the import address table, etc.
- The **Sequence Diagram Link** profiler builds a UML-style diagram of function calls in the profiled application and displays this diagram in Microsoft Word or Microsoft Visio.
- The **Platform Compliance** profiler reports what Windows versions support the API calls in the source.
- The **Exception Trace** profiler monitors the application execution and, if an exception occurs, displays the exception call stack in the Event View panel. Since Exception Trace does not slow down the application execution, it can be very convenient to use if your main goal is tracking down an application exception. Like the Performance profiler, Exception Trace supports the profiling of unmanaged code.
- The **Function Trace** profiler traces the routine calls during the profiler run and logs call stacks for each call. Native-code and managed application profiling is supported (including 64-bit code support). It provides you with comprehensive information on how any routine is invoked, which parameter values are passed to it and some other routine characteristics. This profiler provides an opportunity to process actual call stack data in real-time. Source code modifications are not needed - Function Trace automatically performs actions, that otherwise, could only be done by introducing hundreds of trace-message lines in the source code.
- The **BDE SQL** profiler measures and logs the execution time of SQL queries or SQL stored procedures called through the BDE (Borland Database Engine). Using the Details panel of the profiler you can view the sequence of functions that call a BDE operation.
- The **Load Library Tracer** profiler traces the loading and unloading of dynamic link libraries during the application execution. Using the profiler you can detect which libraries are loaded and unloaded too often (and thus impact the overall application performance) and optimize the use of them.
- The **Unused VCL Units** profiler detects standard VCL and user units that were included in the application but are not used by it. Removing these units will decrease the application size without losing any functionallity.

Preparing an Application for Profiling

How AQtime Profilers Use Metadata and Debug Information

In order to profile your application with AQtime you may need to compile it with debug information. This depends on your application type: *managed* (or .NET-connected) application or *unmanaged* (or native-code, non-.NET) application.

Note: AQtime is not compatible with applications that perform non-standard actions over binary code or over a stack. For more information, see *Unsupported Code* in AQtime help.

Native-code (unmanaged) applications

In order to profile native-code applications with AQtime, **such applications must be compiled with debug information**. Debug info tells AQtime where the routines start and end in memory, what the routine size is (in bytes), where is each routine located in the executable's memory, etc. For complete information on how to compile your application with debug info, see these topics:

- *Compiler Settings for Microsoft Visual C++ 6.0*
- *Compiler Settings for Microsoft Visual Basic*
- *Compiler Settings for Embarcadero Delphi 2010 for Win32*
- *Compiler Settings for CodeGear Delphi 2009 for Win32*
- *Compiler Settings for CodeGear Delphi 2007 for Win32*
- *Compiler Settings for Borland Delphi 2006 for Win32*
- *Compiler Settings for Borland Delphi 2005 for Win32*
- *Compiler Settings for Borland Delphi 3-7*
- *Compiler Settings for Embarcadero C++Builder 2010*
- *Compiler Settings for CodeGear C++Builder 2009*
- *Compiler Settings for CodeGear C++Builder 2007*
- *Compiler Settings for Borland C++Builder 2006*
- *Compiler Settings for Borland C++Builder3-6*
- *Compiler Settings for Borland C++*
- *Compiler Settings for Intel C++*
- *Compiler Settings for GCC*
- *Compiler Settings for Compaq Visual Fortran*

When your application is ready for final delivery, remember to compile it without debug info to reduce the application size.

Managed (.NET) applications

Generally, .NET applications already have all the information necessary for profiling as represented by its metadata. However, metadata holds no information about links between source files and an application's internals (types, classes, members etc.) This limits some AQtime features. For example, the **Editor** panel does not display the source code for the routine selected in the **Report** panel. Another limitation is that you cannot profile routines at the **line level** (see *Profiling Levels*).

The above limitations are eliminated when you include debug information to your application executable and AQtime will use the debug information as an addition to the existing .NET metadata information. To learn how to add debug information to your .NET applications, see these topics:

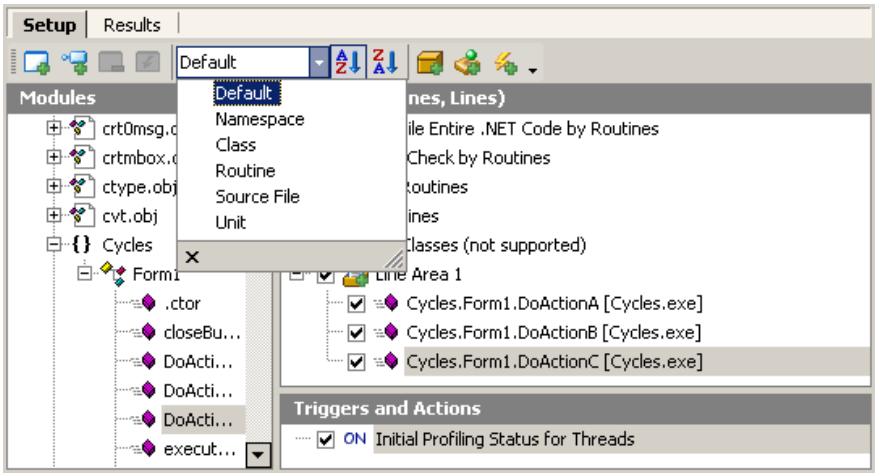
- *Compiler Settings for ASP.NET*
- *Compiler Settings for Microsoft Visual C# 2005 and 2008*
- *Compiler Settings for Microsoft Visual C# .NET*
- *Compiler Settings for Microsoft Visual J# 2005 and 2008*
- *Compiler Settings for Microsoft Visual J# .NET*
- *Compiler Settings for Microsoft Visual Basic 2005 and 2008*
- *Compiler Settings for Microsoft Visual Basic .NET*
- *Compiler Settings for Microsoft Visual C++ 2005 and 2008*
- *Compiler Settings for Microsoft Visual C++ .NET*
- *Compiler Settings for Borland C#Builder 2006*
- *Compiler Settings for Borland C#Builder*
- *Compiler Settings for Borland Delphi 2006 for .NET*
- *Compiler Settings for Borland Delphi 2005 for .NET*
- *Compiler Settings for Borland Delphi 8 (Delphi for .NET)*

When your .NET application is ready for final delivery, remember to compile it without debug info to reduce the application size.

Note: If your managed application includes portions of unmanaged code, you should compile it with debug information in order for AQtime to be able to profile the unmanaged code.

Let's reiterate that AQtime profiles routines at the line level only if the application was compiled with debug information. In addition to Line Level profiling, debug information provides other benefits:

- The Editor panel displays the source code when you double-click a routine in the **Report** or **Details** panels or the **Modules** pane of the **Setup** panel.
- The Setup panel lets you navigate through the application not only by namespace and class but also by file:



- For some profilers, the Report and Details panels will show the **Source File** and **Source Line** columns. These columns are filled with data obtained exclusively from the application's debug info. They help you locate a routine in your source quickly.

Use the Symbols Options dialog to enable or disable debug readers, which AQtime uses to parse debug information and metadata. In the dialog, you can also specify the search paths for debug info files that are generated separately from the executables.

Compiler Settings for ASP.NET

This topic explains how to prepare ASP.NET applications before profiling them in AQtime.

To add debug information to your ASP.NET application, follow these steps:

1. Compile your ASP.NET application with debug information. The actions you should perform at this step depend on the tool you are using to create and compile your application. For more details, see the topics listed below:
   - *Compiler Settings for Microsoft Visual C# 2005*
   - *Compiler Settings for Microsoft Visual C# .NET*
   - *Compiler Settings for Microsoft Visual Basic 2005*
   - *Compiler Settings for Microsoft Visual Basic .NET*
   - *Compiler Settings for Microsoft Visual J# 2005*
   - *Compiler Settings for Microsoft Visual J# .NET*
   - *Compiler Settings for Borland C#Builder 2006*
   - *Compiler Settings for Borland C#Builder*
   - *Compiler Settings for Borland Delphi 2006 for .NET*
   - *Compiler Settings for Borland Delphi 2005 for .NET*
   - *Compiler Settings for Borland Delphi 8 (Delphi for .NET)*
2. Create the web.config file in your project folder (if it is not there yet) and add the `<compilation debug="true" />` line to this file.

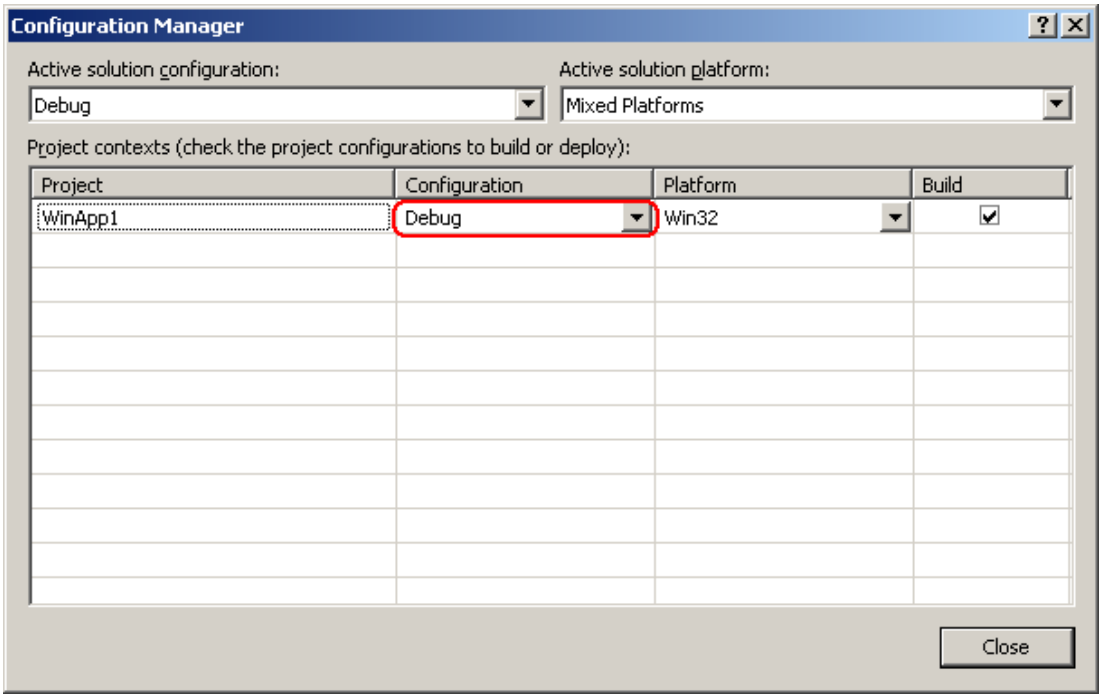
When your application is ready for final delivery, remember to compile it without debug information to reduce the overall size of the application.

Compiler Settings for Microsoft Visual C# 2005 and 2008

This topic explains how to prepare applications created with Microsoft Visual C# 2005 or Visual C# 2008 for AQtime. To learn how to prepare applications created with Visual C# .NET, see *Compiler Settings for Microsoft C# .NET*.

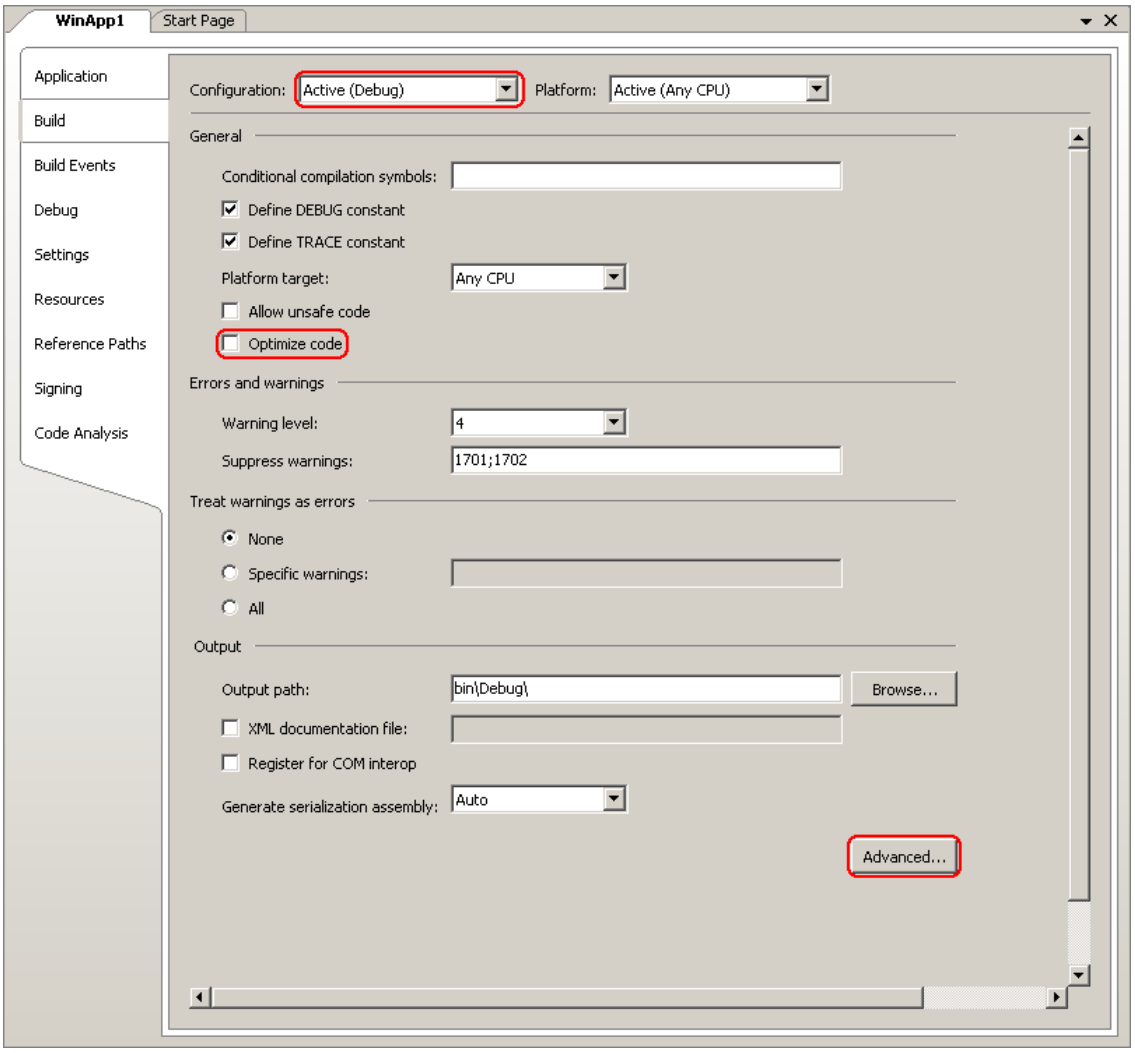
To add debug information to your Visual C# 2005 or Visual C# 2008 application, follow these steps:

1. Open your project in Visual Studio 2005 or Visual Studio 2008.
2. Select **Build | Configuration Manager** from the main menu. This will open the **Configuration Manager** dialog. Select the *Debug* configuration for your project:

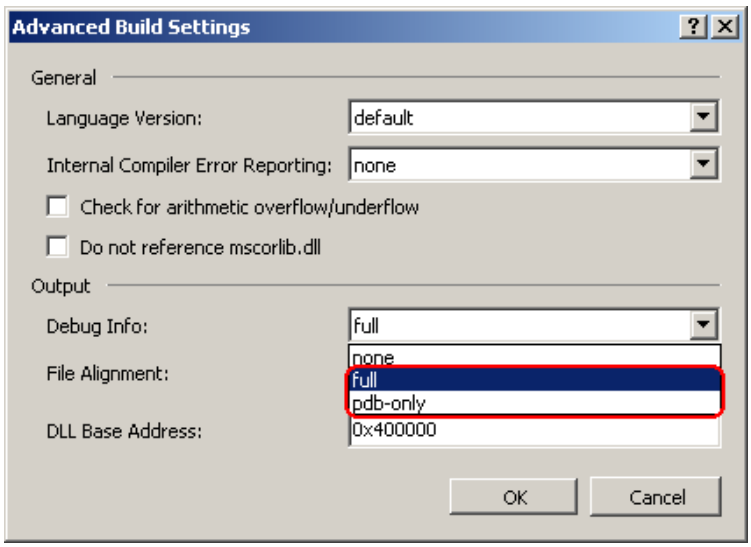


Close the dialog.

3. Right-click the project in the **Solution Explorer** and select **Properties** from the context menu. This will open the **Property Pages** dialog.
4. In this dialog, open the **Build** page and select *Active (Debug)* from the **Configuration** dropdown list at the top of the dialog. Then, on the same **Build** page, disable the **Optimize code** checkbox and click the **Advanced** button.

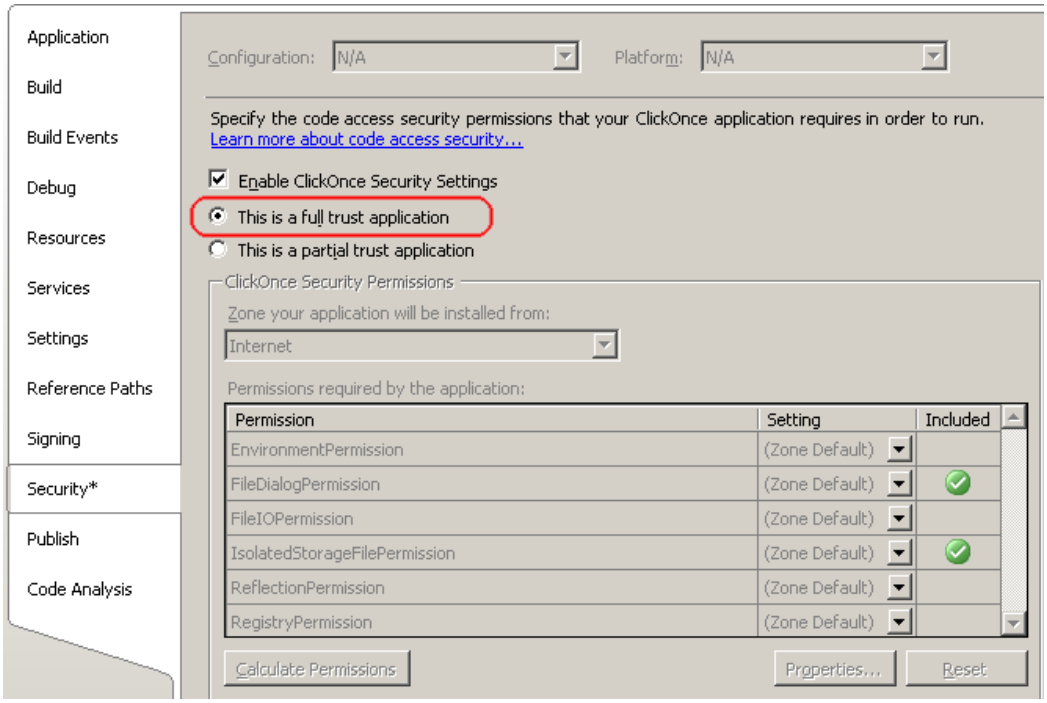


5. In the resulting **Advanced Build Settings** dialog, select either *full* or *pdb-only* in the **Debug Info** dropdown list box.



6. Click **OK** to close the dialog.

7. If your application is a WPF Browser application (XBAP), then you should change the security settings: switch to the **Security** page and enable the **This is a full trust application** setting:



8. Save the changes in the project. Recompile the application.

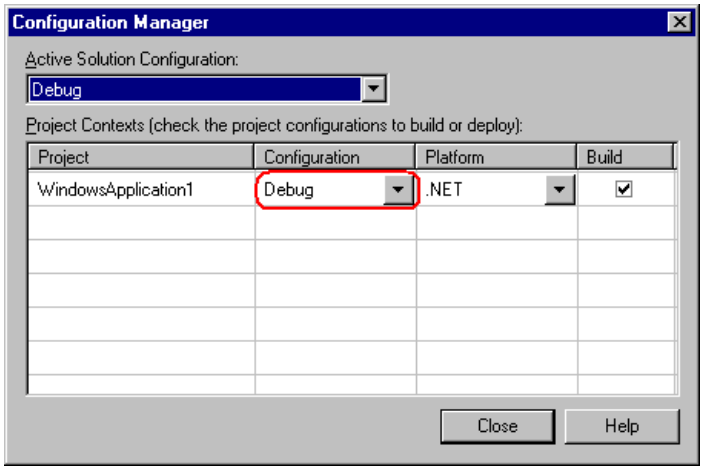
When your application is ready for final delivery, remember to compile it without debug information to reduce the overall size of the application.

Compiler Settings for Microsoft Visual C# .NET

This topic explains how to prepare applications created with Microsoft Visual C# .NET for AQtime. To learn how to prepare applications created with Visual C# 2005, see *Compiler Settings for Microsoft C# 2005*.

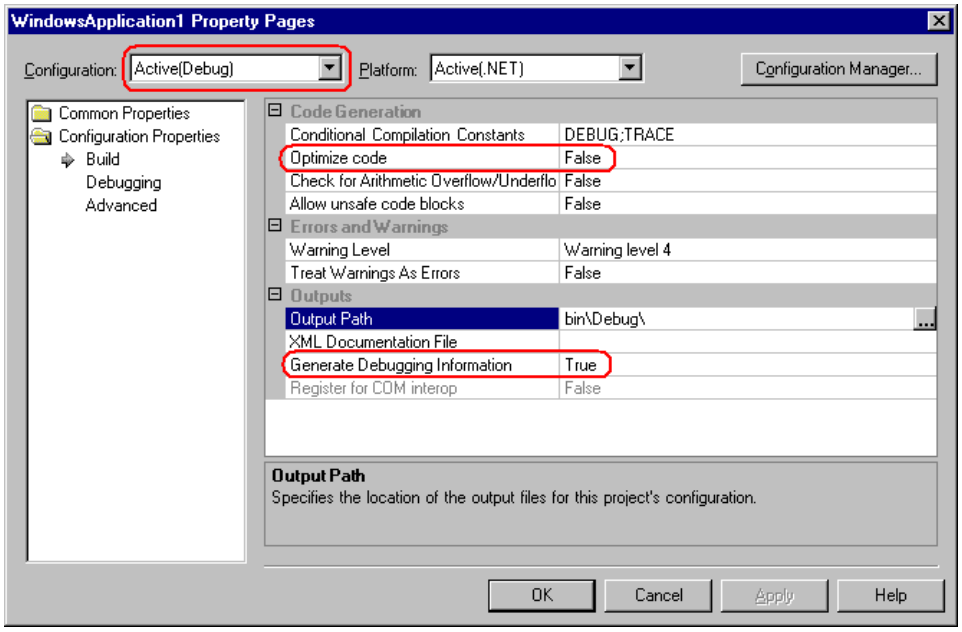
To add debug information to your Visual C# .NET application, follow these steps:

1. Open your project in Visual Studio .NET.
2. Select **Build | Configuration Manager** from the main menu. This will open the **Configuration Manager** dialog. Select the *Debug* configuration for your project:



Close the dialog.

3. Right-click the project in the Solution Explorer and select **Properties** from the context menu. This will open the **Property Pages** dialog.
4. In the dialog, open the **Configuration Properties | Build** page and select *Active (Debug)* from the **Configuration** dropdown list at the top of the dialog. Then, in the **Configuration Properties | Build** page, turn on the **Generate debugging information** option and disable **Optimize Code**.



5. Press **OK** to save settings. Recompile the application.

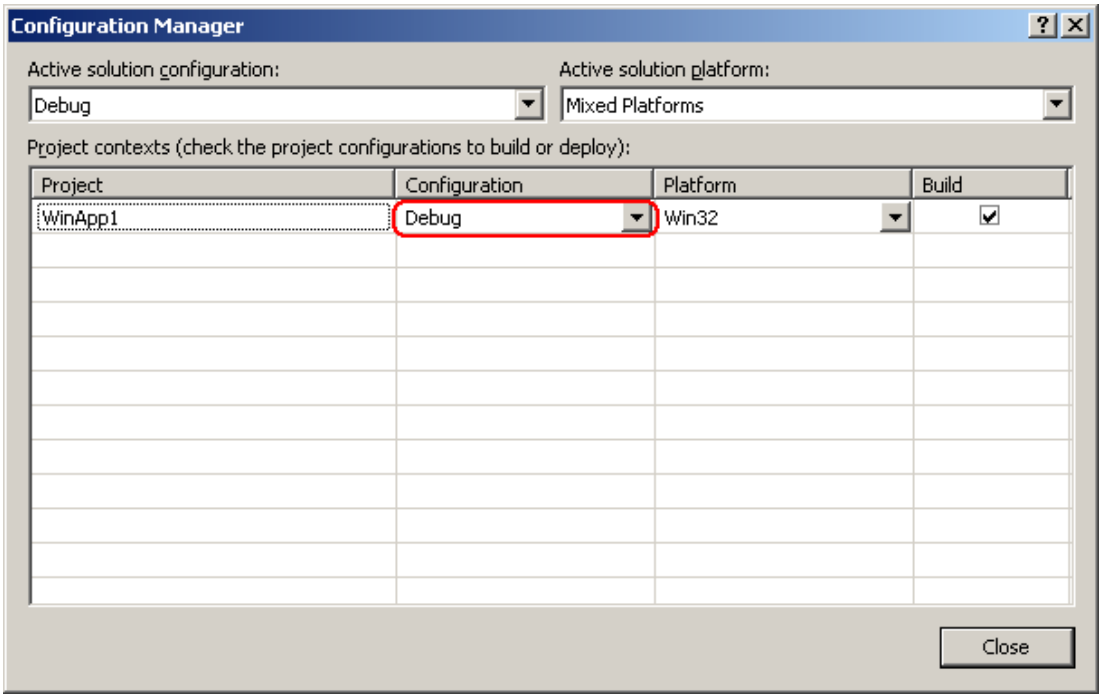
When your application is ready for final delivery, remember to compile it without debug information to reduce overall application size.

Compiler Settings for Microsoft Visual J# 2005 and 2008

This topic explains how to prepare applications created with Visual J# 2005 or Visual J# 2008 for AQtime. To learn how to prepare applications created with Visual J# .NET, see *Compiler Settings for Microsoft Visual J# .NET*.

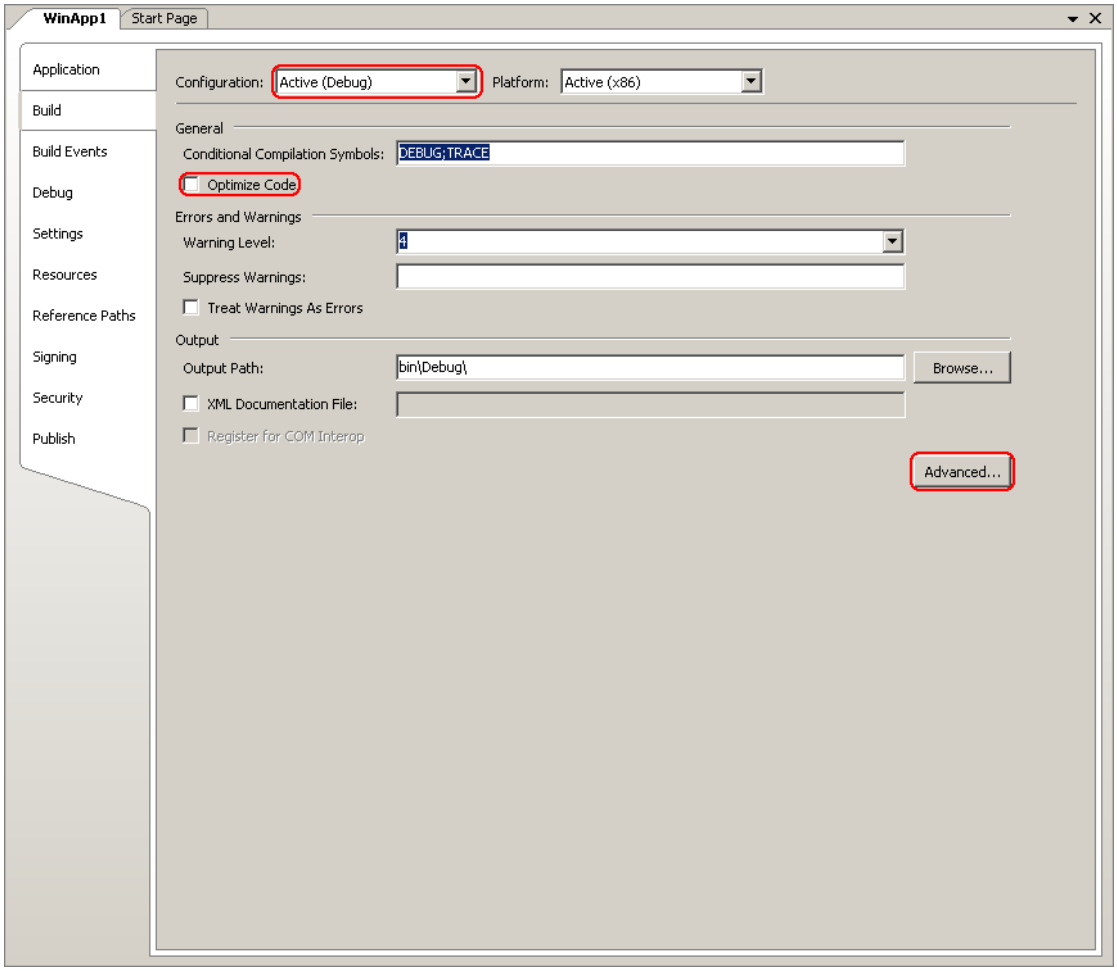
To add debug information to your Visual J# 2005 or Visual J# 2008 application, follow these steps:

1. Open your project in Visual Studio 2005 or Visual Studio 2008.
2. Select **Build | Configuration Manager** from the main menu. This will open the **Configuration Manager** dialog. Select the *Debug* configuration for your project:

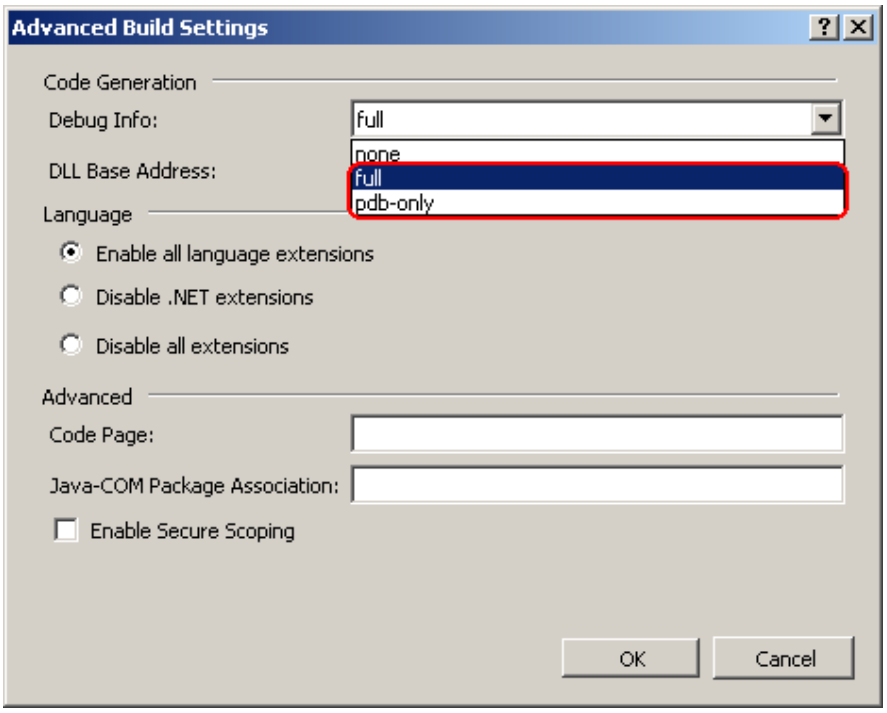


Close the dialog.

3. Right-click the project in the **Solution Explorer** and select **Properties** from the context menu. This will open the **Property Pages** dialog.
4. In this dialog, open the **Build** page and select *Active (Debug)* from the **Configuration** dropdown list at the top of the dialog. Then, on the same **Build** page, disable the **Optimize Code** checkbox and click the **Advanced** button.



5. In the resulting **Advanced Build Settings** dialog, select either *full* or *pdb-only* in the **Debug Info** dropdown list box.



6. Click **OK** to close the dialog.
7. Save the changes in the project. Recompile the application.

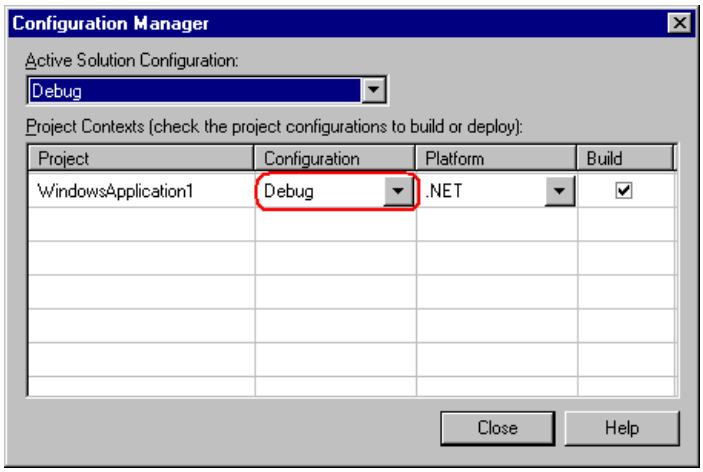
When your application is ready for final delivery, remember to compile it without debug information to reduce the overall size of the application.

Compiler Settings for Microsoft Visual J# .NET

This topic explains how to prepare applications created with Visual J# .NET for AQtime. To learn how to prepare applications created with Visual J# 2005, see *Compiler Settings for Microsoft Visual J# 2005*.

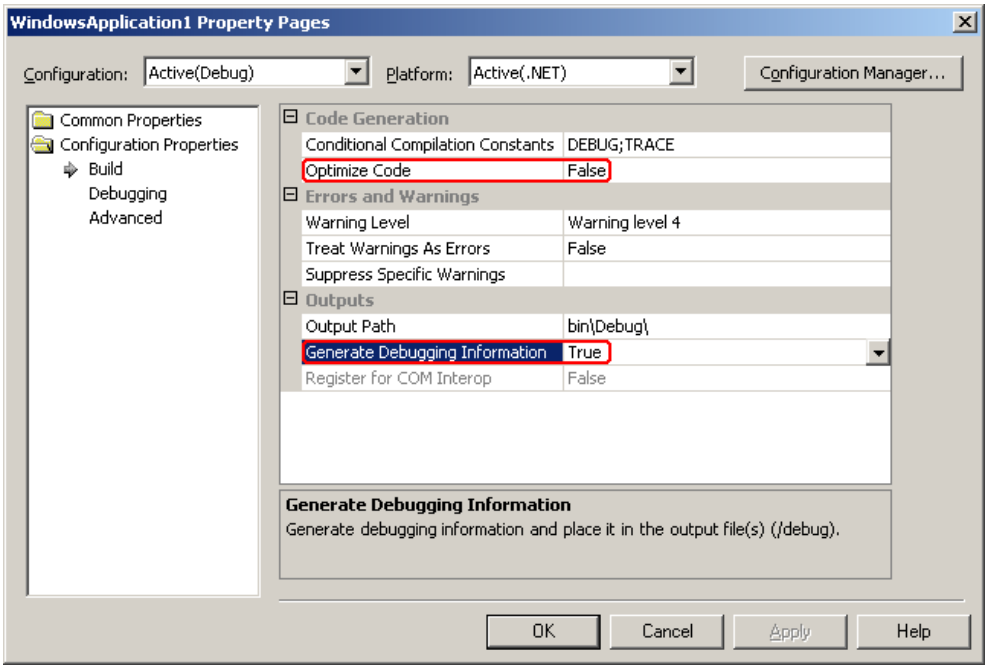
To add debug information to your Visual J# .NET application, follow these steps:

1. Open your project in Visual Studio .NET.
2. Select **Build | Configuration Manager** from the main menu. This will open the **Configuration Manager** dialog. Select the *Debug* configuration for your project:



Close the dialog.

3. Right-click the project in the Solution Explorer and select **Properties** from the context menu. This will open the **Property Pages** dialog.
4. In the dialog, open the **Configuration Properties | Build** page and select *Active (Debug)* from the **Configuration** dropdown list at the top of the dialog. Then, in the **Configuration Properties | Build** page, turn on the **Generate debugging information** option and disable **Optimize Code**.



5. Press **OK** to save settings. Recompile the application.

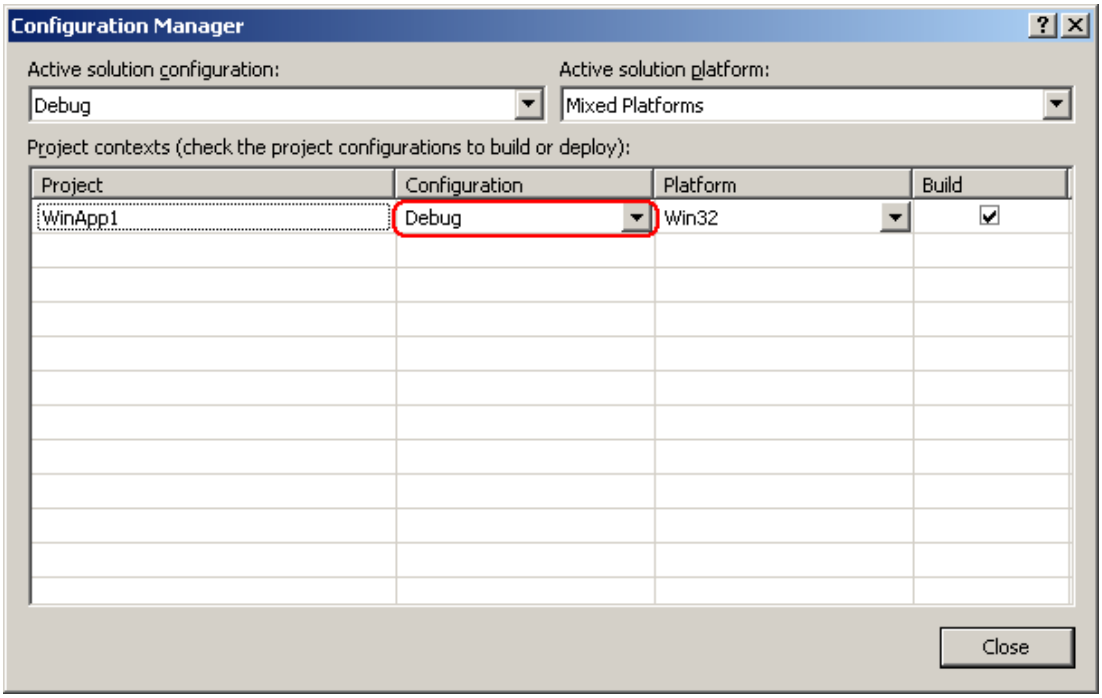
When your application is ready for final delivery, remember to compile it without debug information to reduce overall application size.

Compiler Settings for Microsoft Visual Basic 2005 and 2008

This topic explains how to prepare applications that were created with Microsoft Visual Basic 2005 or Visual Basic 2008 for AQtime. To learn how to prepare applications created with Visual Basic 6.0 or Visual Basic .NET, see *Compiler Settings for Microsoft Visual Basic* and *Compiler Settings for Microsoft Visual Basic .NET* respectively.

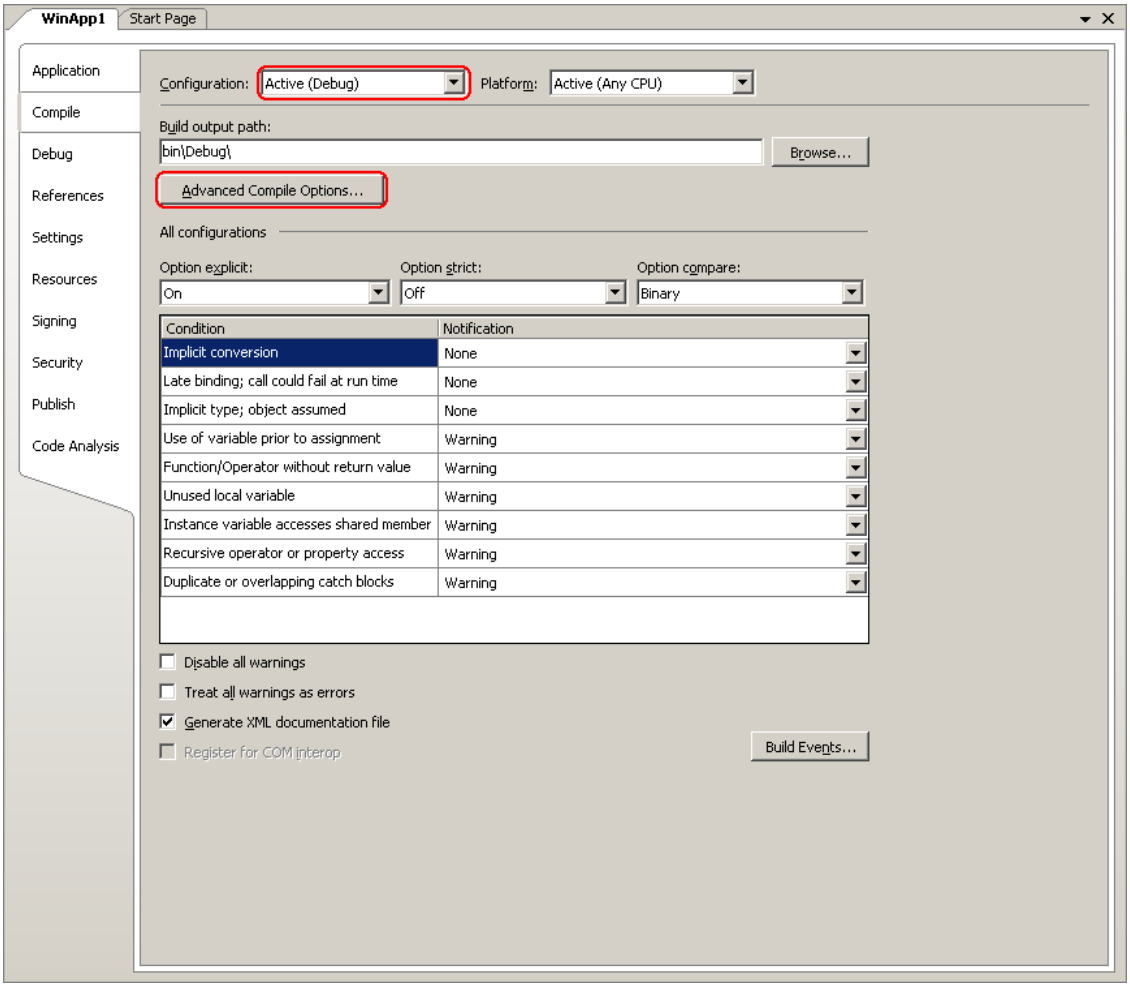
To add debug information to your Visual Basic 2005 or Visual Basic 2008 application, follow these steps:

1. Open your project in Visual Studio 2005 or Visual Studio 2008.
2. Select **Build | Configuration Manager** from the main menu. This will open the **Configuration Manager** dialog. Select the *Debug* configuration for your project:

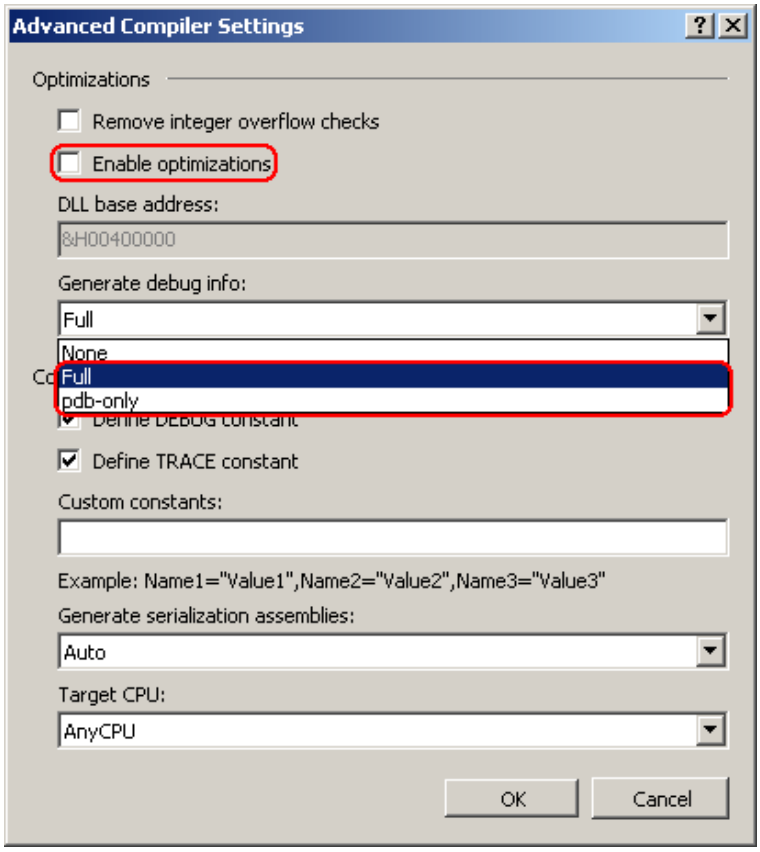


Close the dialog.

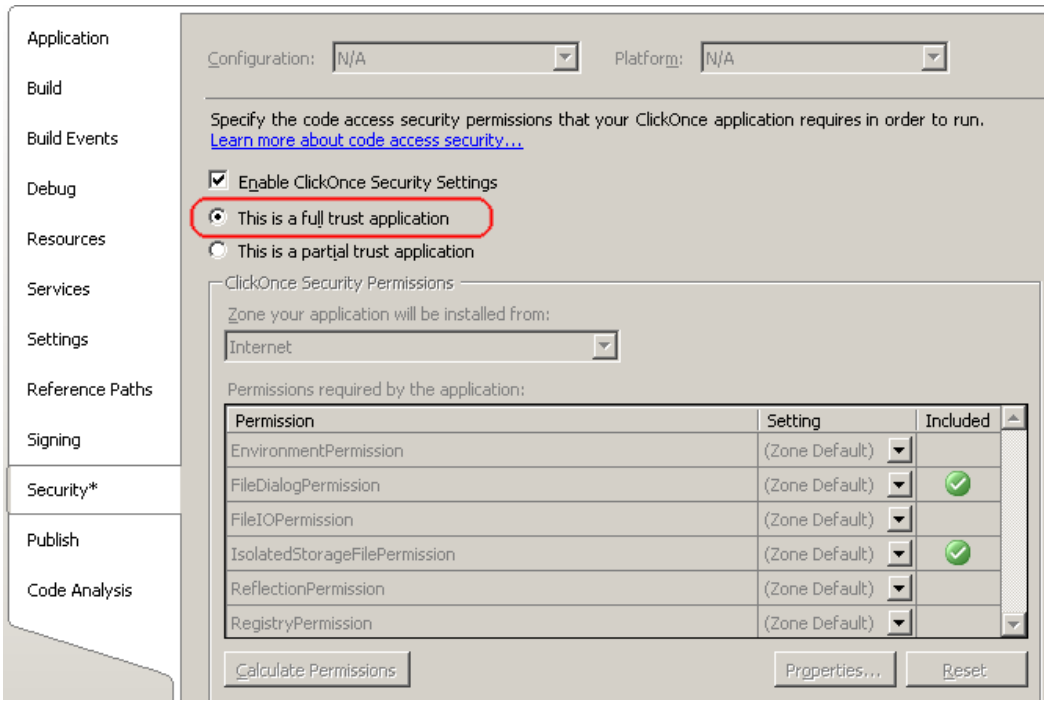
3. Right-click the project in the **Solution Explorer** and select **Properties** from the context menu. This will open the **Property Pages** dialog.
4. In the dialog, open the **Build** page and select *Active (Debug)* from the **Configuration** dropdown list at the top of the dialog. Then on the same **Build** page, click the **Advanced Compile Options** button.



5. In the resulting **Advanced Build Settings** dialog, select either *Full* or *pdb-only* in the **Generate debug info** dropdown list box and turn off the **Enable optimizations** checkbox.



6. Click **OK** to close the dialog.
7. If your application is a WPF Browser application (XBAP), then you should change the security settings: switch to the **Security** page and enable the **This is a full trust application** setting:



8. Save the changes in the project and recompile the application.

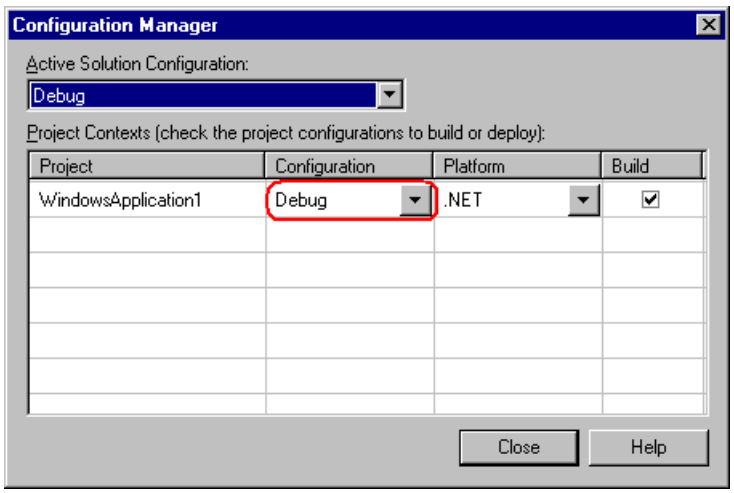
When your application is ready for final delivery, remember to compile it without debug information to reduce the overall size of the application.

Compiler Settings for Microsoft Visual Basic .NET

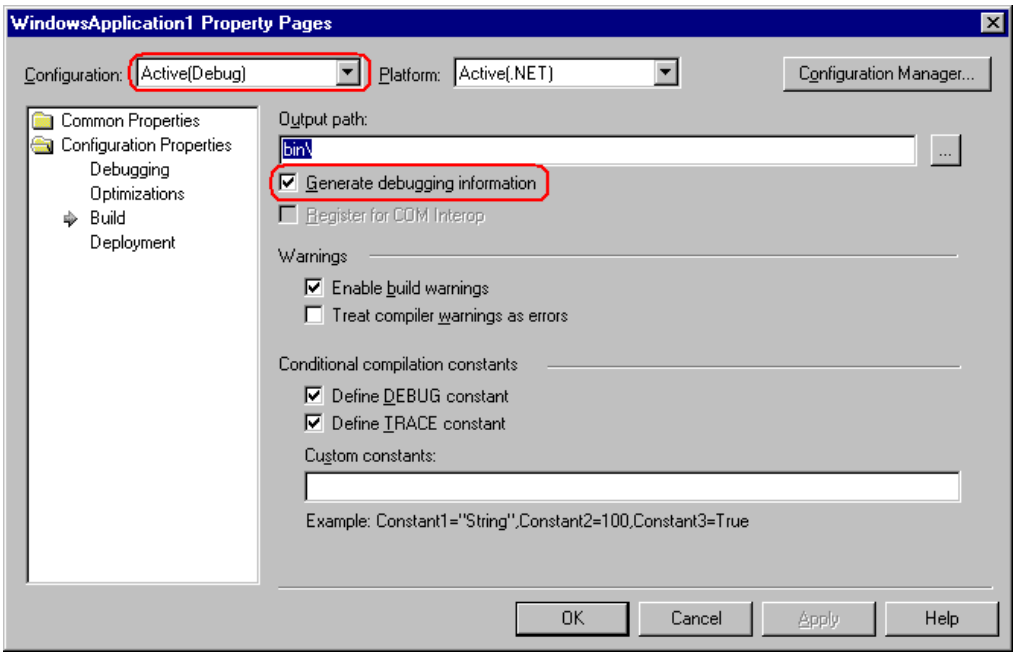
This topic explains how to prepare applications that were created with Microsoft Visual Basic .NET for AQtime. To learn how to prepare applications created with Visual Basic 6.0, see *Compiler Settings for Microsoft Visual Basic*. To learn how to prepare applications created with Visual Basic 2005, see *Compiler Settings for Microsoft Visual Basic 2005*.

To add debug information to your Visual Basic .NET application, follow these steps:

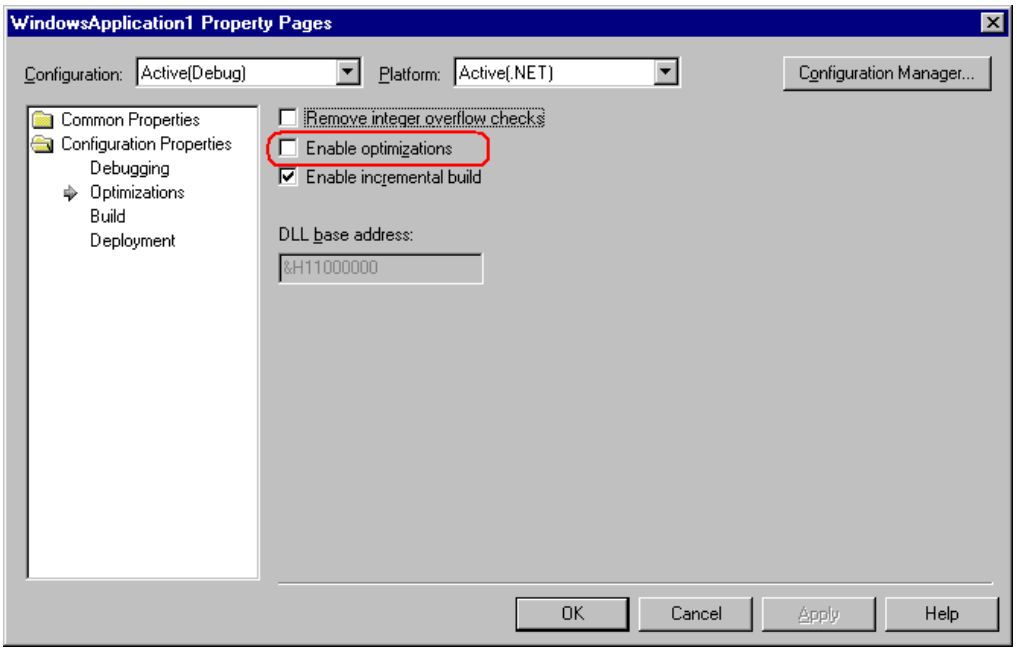
1. Open your project in Visual Studio .NET.
2. Select **Build | Configuration Manager** from the main menu. This will open the **Configuration Manager** dialog. Select the *Debug* configuration for your project:



   Close the dialog.
3. Right-click the project in the Solution Explorer and select **Properties** from the context menu. This will open the **Property Pages** dialog.
4. In the dialog,
   - Open the **Configuration Properties | Build** page and select *Active (Debug)* from the **Configuration** dropdown list at the top of the dialog.
   - In the **Configuration Properties | Build** page enable the **Generate debugging information** option.



- Switch to the **Configuration Properties | Optimizations** page and turn off the **Enable Optimizations** option.



5. Press **OK** to save settings and recompile the application.

When your application is ready for final delivery, remember to compile it without debug information to reduce overall application size.

Compiler Settings for Microsoft Visual C++ 2005 and 2008

This topic explains how to prepare applications created with Visual C++ 2005 and Visual C++ 2008 for AQtime. To learn how to prepare applications created with Visual C++ 7.x, see *Compiler Settings for*

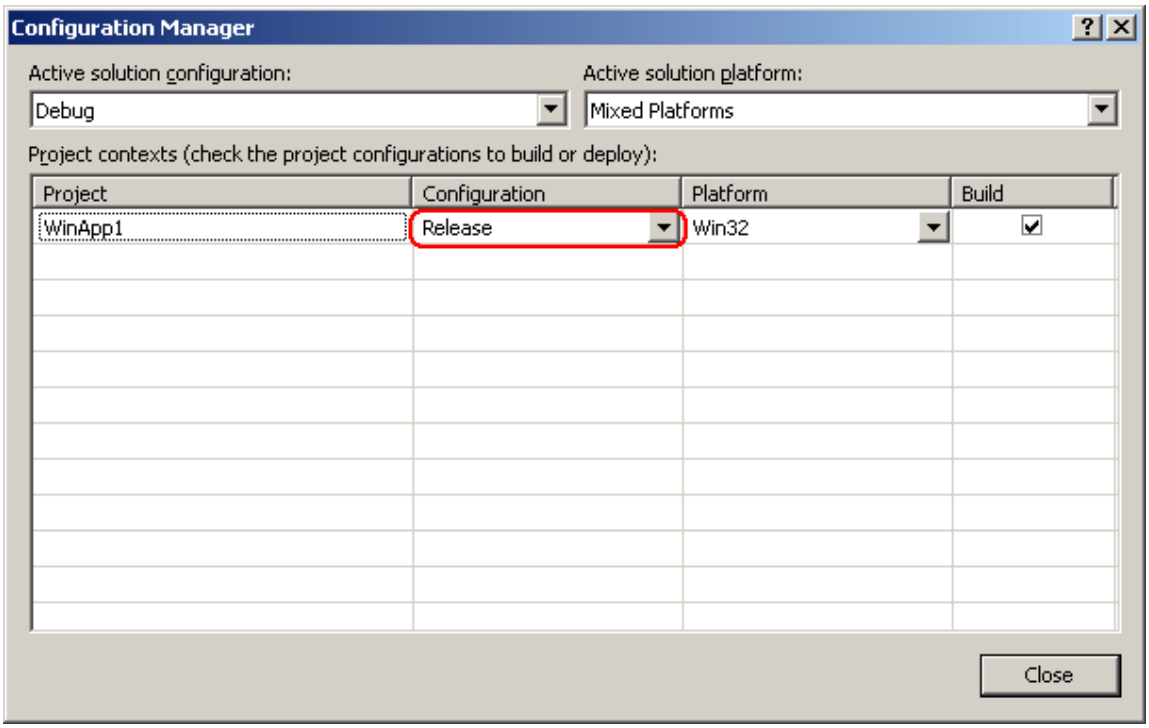
Microsoft Visual C++ .NET. To learn how to prepare applications created with Visual C++ 6.0 or earlier, see *Compiler Settings for Microsoft Visual C++ 6.0*.

Note that default settings created by Microsoft Visual Studio 2005 for a new Visual C++ 2005 or 2008 application already contain the necessary information for generating debug information. So, if you do not change the compiler settings, you can compile and profile your Visual C++ 2005 (or 2008) application as is. However, we recommend that you change the active configuration to *Release* if you are going to profile with the **Performance** profiler. The *Release* configuration is used to build the final version of the application, so you should search for performance bottlenecks in versions compiled with the *Release* configuration, but not with the *Debug* configuration.

If you are not sure whether your application's compiler settings were changed, you can follow the steps described in this topic to make sure that your application is compiled with debug information.

To add debug information to your Visual C++ 2005 or Visual C++ 2008 application, follow these steps:

1. Open your project in Visual Studio 2005 or Visual Studio 2008.
2. Select **Build | Configuration Manager** from the main menu. This will open the **Configuration Manager** dialog. Select the *Release* configuration for your project:

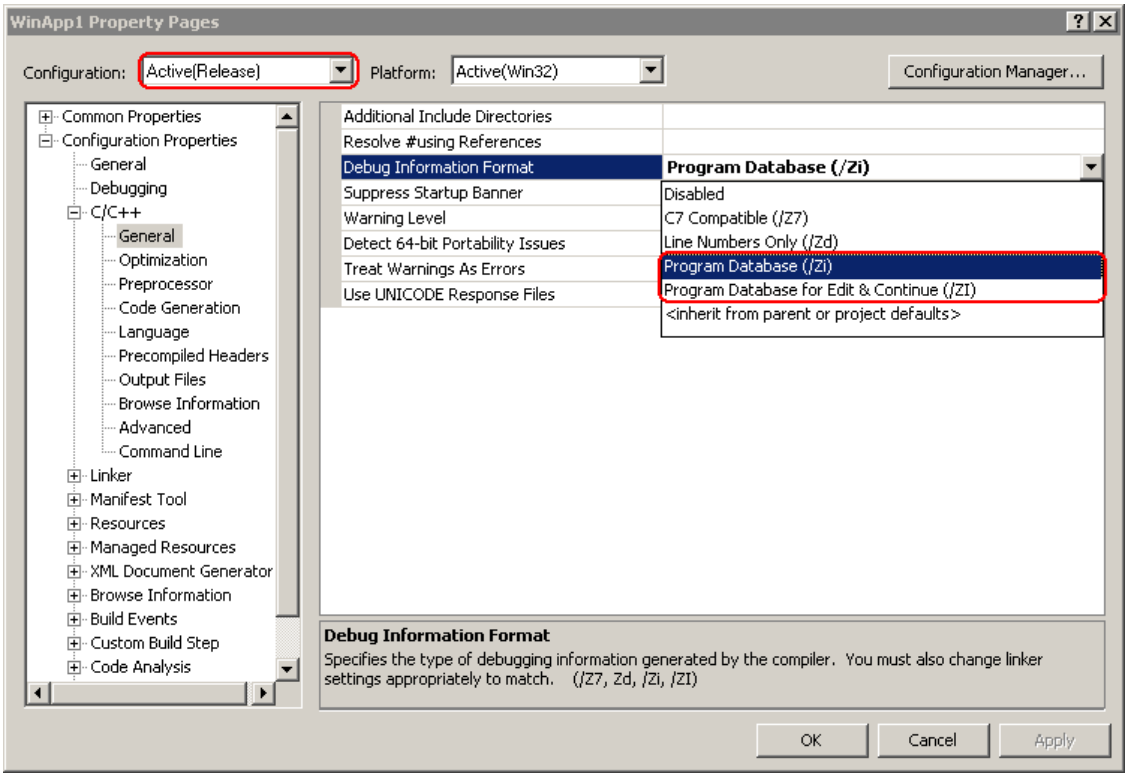


Close the dialog.

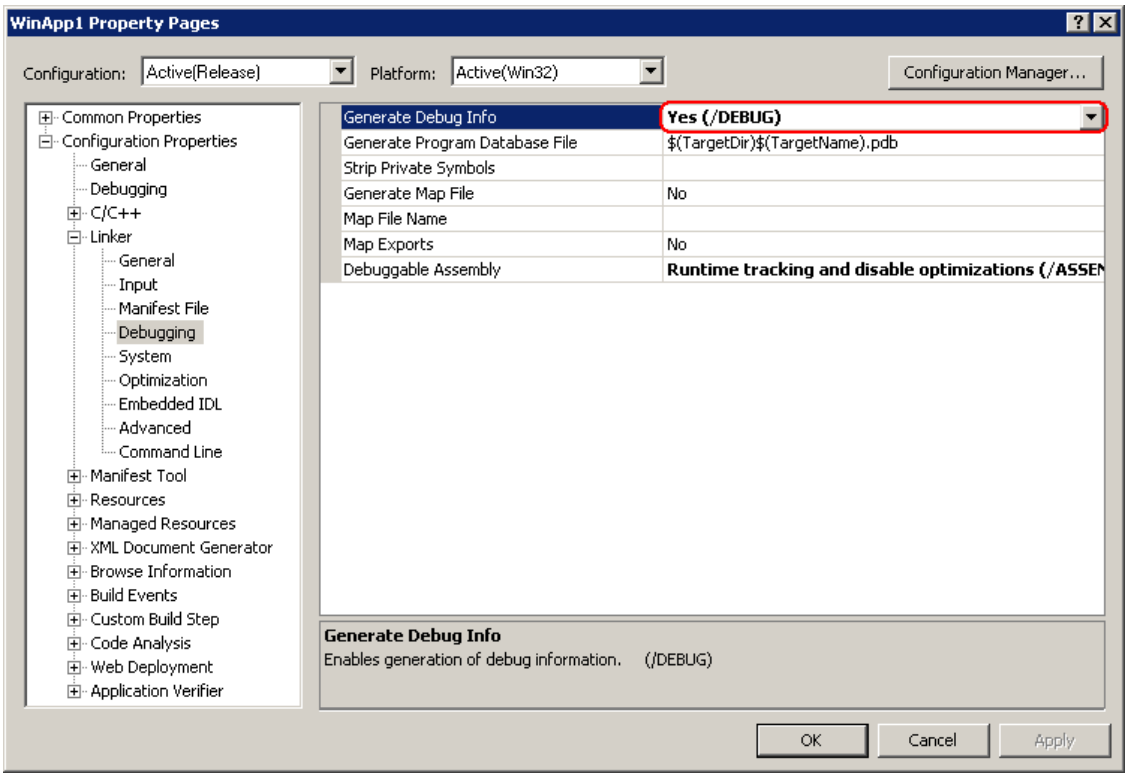
Note: Of course, you can profile your application in any configuration, not just in the *Release* one. We recommend that you choose this configuration if you are going to use the Performance profiler (see above).

3. Right-click the project in the Solution Explorer and select **Properties** from the context menu. This will call the **Property Pages** dialog.
4. In the dialog,
   - Select *Active (Release)* from the **Configuration** dropdown list.

- Switch to the **Configuration Properties | C/C++ | General** page and set **Debug Information Format** to *Program Database (/Zi)* or *Program Database for Edit & Continue (/ZI)*. For more information on these options review Visual Studio 2005 Help.



- Open the **Configuration Properties | Linker | Debugging** property page and set the **Generate Debug Info** option to *Yes (/DEBUG)*.



5. Click **OK** to save the settings and then rebuild the application. Once finished, your application will be ready for profiling in AQtime.

Before opening the application in AQtime, make sure that the output directory of your application includes the .pdb file that you have created. In addition, this directory should contain the vcX0.pdb file, where X is the major version number of the Visual C++ compiler, e.g. for Visual C++ 8.0 this file is called vc80.pdb. vcX0.pdb contains part of your application's debugging information, which may be needed to profile the entire application in AQtime correctly. It is possible that the vc80.pdb file is not generated by Visual Studio and this will not cause any problems with profiling. If you are profiling an ActiveX control or a COM server, do not forget to register its "debug" version in the system (See *Profiling COM Applications*).

By default, the .pdb debug info file resides in the same folder where the profiled executable or DLL is located. When you open your module in AQtime, it searches for the debug info file in this default lcoation. If you copy the debug info file to another location, then specify this location in the **Symbols Path** list of the Symbols Options dialog (this list contains the search paths). See *Specifying Path to Debug Info Files* for more information.

When your application is ready for final delivery, remember to compile it without debug information to reduce the overall size of the application.

Compiler Settings for Microsoft Visual C++ .NET

This topic explains how to prepare applications created with Visual C++ 7.x for AQtime. To learn how to prepare applications created with Visual C++ 2005 (8.0), see *Compiler Settings for Microsoft Visual C++ 2005*. To learn how to prepare applications created with Visual C++ 6.0 or earlier, see *Compiler Settings for Microsoft Visual C++ 6.0*.

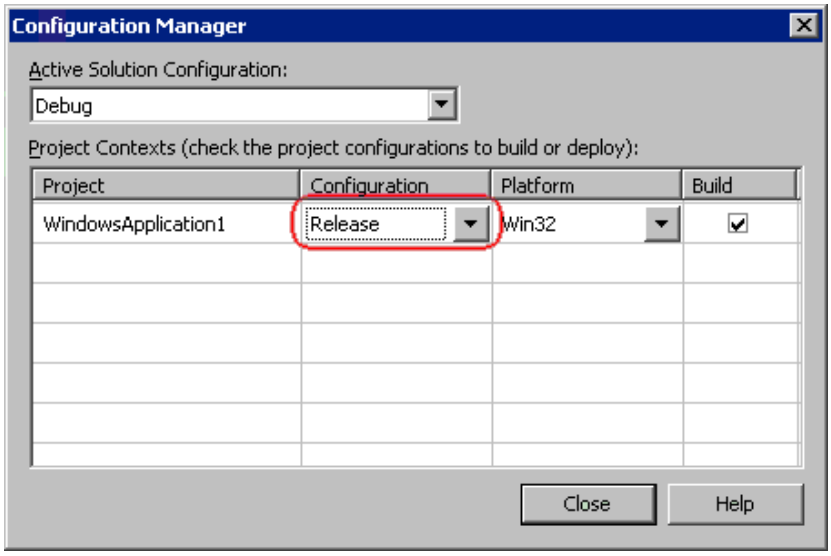
Note that default settings created by Microsoft Visual Studio for a new Visual C++ .NET application already hold all necessary information for generating debug information. So, if you do not change the compiler settings, you can compile and profile your Visual C++ .NET application as is. The only thing we would

recommend is to change the active configuration to *Release* if you are going to profile with the Performance profiler. The *Release* configuration is used to build the final version of the application, so you should search for performance bottlenecks in versions compiled with the *Release* configuration, but not with the *Debug* configuration.

If you are not sure whether your application's compiler settings were changed, you can follow the steps described in this topic to make certain that your application is compiled with debug information.

To add debug information to your Visual C++ .NET application, follow these steps:

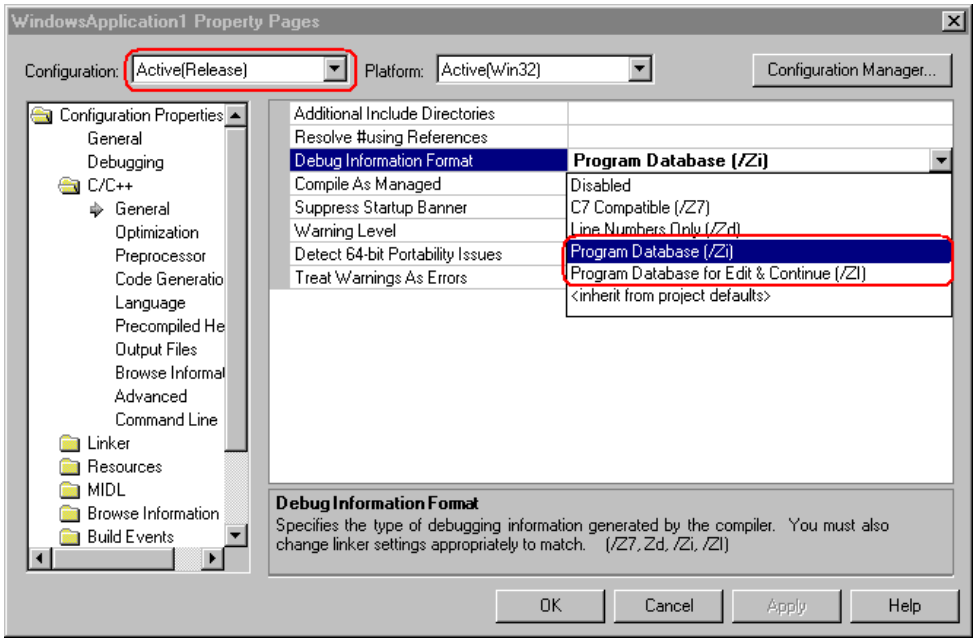
1. Open your project in Visual Studio .NET.
2. Select **Build | Configuration Manager** from the main menu. This will open the **Configuration Manager** dialog. Select the *Release* configuration for your project:



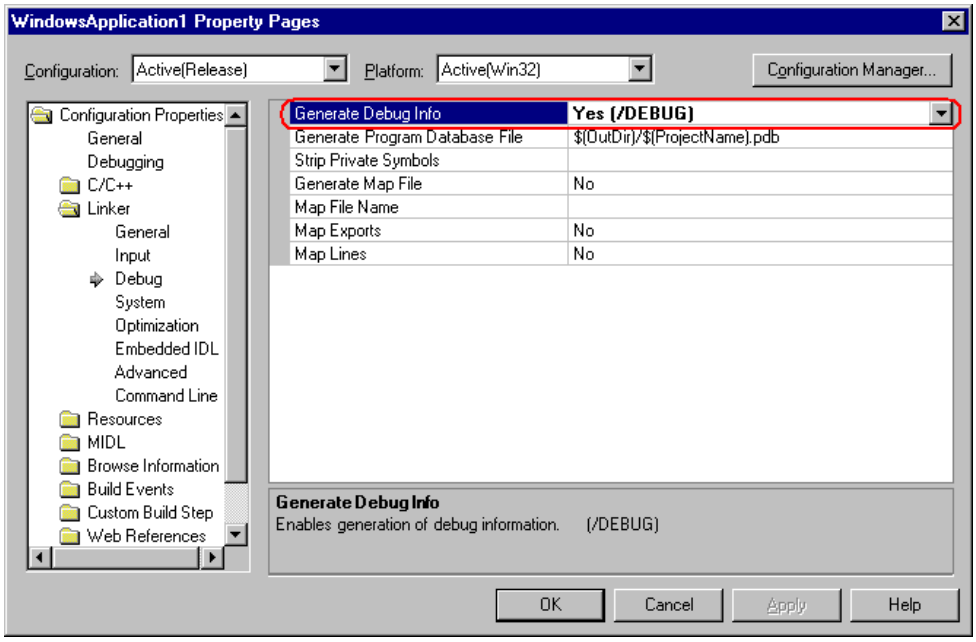
   Close the dialog.

   Note: Of course, you can profile your application in any configuration, not just in the *Release* one. We recommend that you choose this configuration if you are going to use the Performance profiler (see above).

3. Right-click the project in the Solution Explorer and select **Properties** from the context menu. This will call the **Property Pages** dialog.
4. In the dialog,
   - Select *Active (Release)* from the **Configuration** dropdown list.
   - Switch to the **Configuration Properties | C\C++ | General** page and set **Debug Information Format** either to *Program Database (/Zi)* or to *Program Database for Edit & Continue (/ZI)*. For more information on these options review Visual Studio.NET Help.



- Open the **Configuration Properties | Linker | Debug** property page and set the **Generate Debug Info** option to *Yes (/DEBUG)*.



5. Click **OK** to save settings and then rebuild the application. After that it will be ready for profiling in AQtime.

Before opening the application in AQtime, make sure that the output directory of your application includes the .pdb file you have created. In addition, this directory should contain the vcX0.pdb file, where *X* is the major version number of the Visual C++ compiler, e.g. for Visual C++ 7.0 this file is called vc70.pdb. vcX0.pdb holds part of your application's debugging information, which may be needed to profile the entire application in AQtime correctly. It is possible that the vc70.pdb file is not generated by Visual Studio and this will not cause any problems with profiling. If you are profiling an ActiveX control or a COM server, do not forget to register its "debug" version in the system (See *Profiling COM Applications*).

© 2010 AutomatedQA Corp.

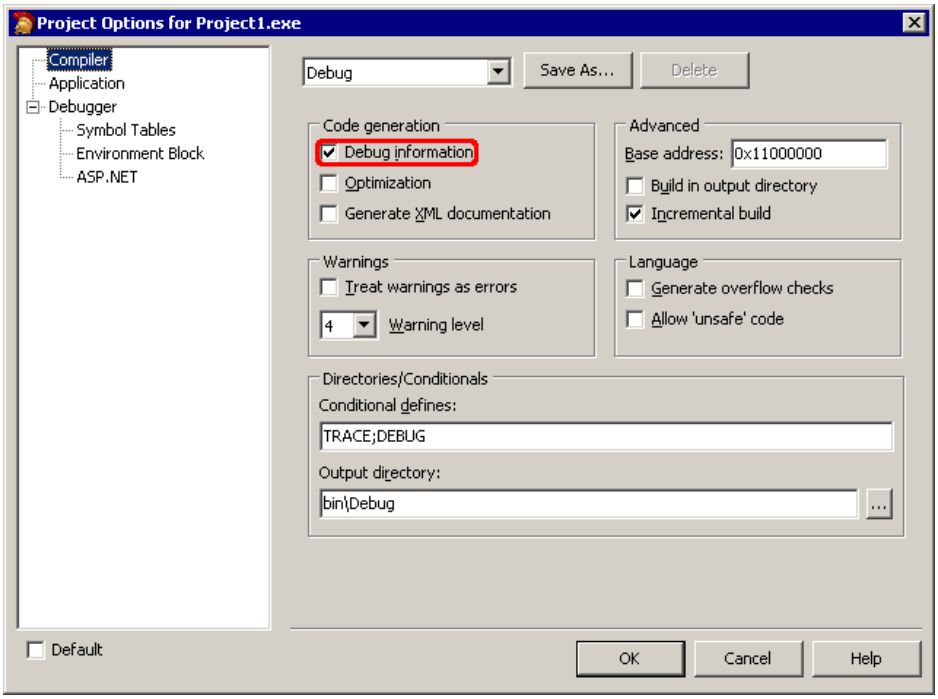
When your application is ready for final delivery, remember to compile it without debug information to reduce overall application size.

Compiler Settings for Borland C#Builder 2006

This topic explains how to prepare applications created with Borland C#Builder 2006 for AQtime. To learn how to prepare applications created with earlier versions of Borland C#Builder, see *Compiler Settings for Borland C#Builder*.

To compile your application with debug info in Borland C#Builder 2006, follow these steps:

1. Open your application in Borland C#Builder 2006.
2. Select **Project | Options** from C#Builder's main menu. This will open the **Project Options** dialog.
3. In the dialog, select the **Compiler** category and check the **Debug information** option:



   Close the dialog.
4. Click **OK** to save changes.
5. Recompile your application.

When your application is ready for final delivery, remember to compile it without debug information to reduce the overall application size.

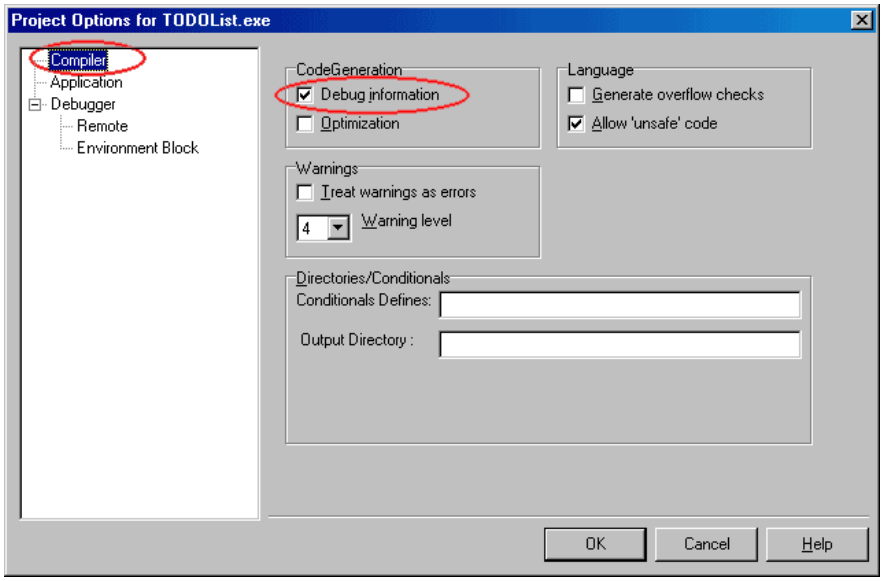
Compiler Settings for Borland C#Builder

This topic explains how to prepare applications created with Borland C#Builder for AQtime. To learn how to prepare applications created with Borland C#Builder 2006, see *Compiler Settings for Borland C#Builder 2006*.

To compile your application with debug info in Borland C#Builder, follow these steps:

1. Open your application in Borland C#Builder.
2. Select **Project | Options** from C#Builder's main menu. This will open the **Project Options** dialog.

3. In the dialog, select the **Compiler** category and check the **Debug information** option:



   Close the dialog.

4. Click **OK** to save changes and recompile your application.

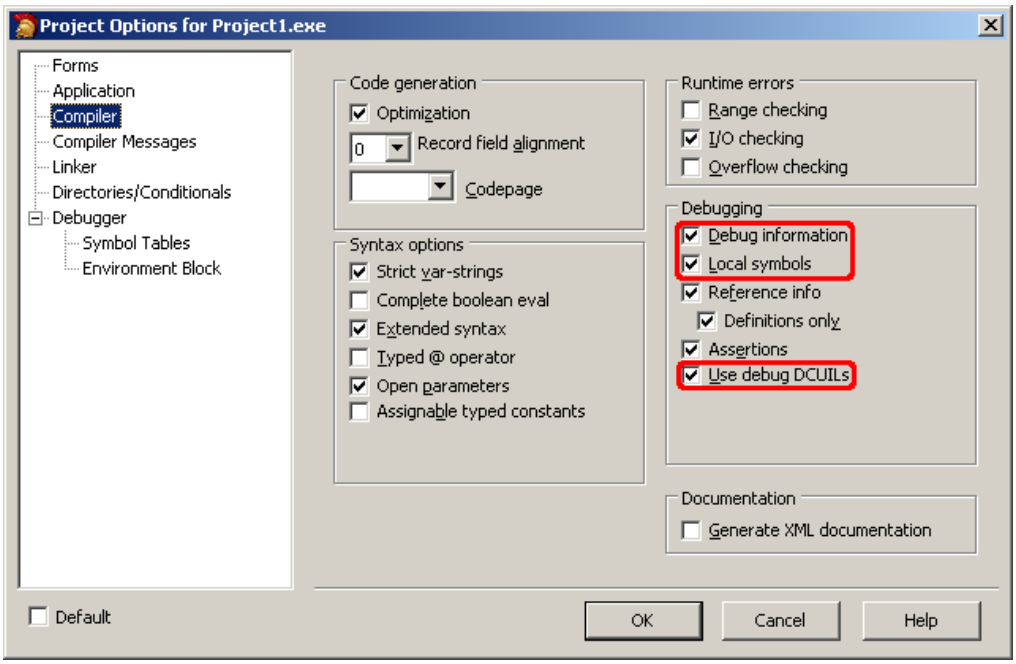
When your application is ready for final delivery, remember to compile it without debug information to reduce overall application size.

Compiler Settings for Borland Delphi 2006 for .NET

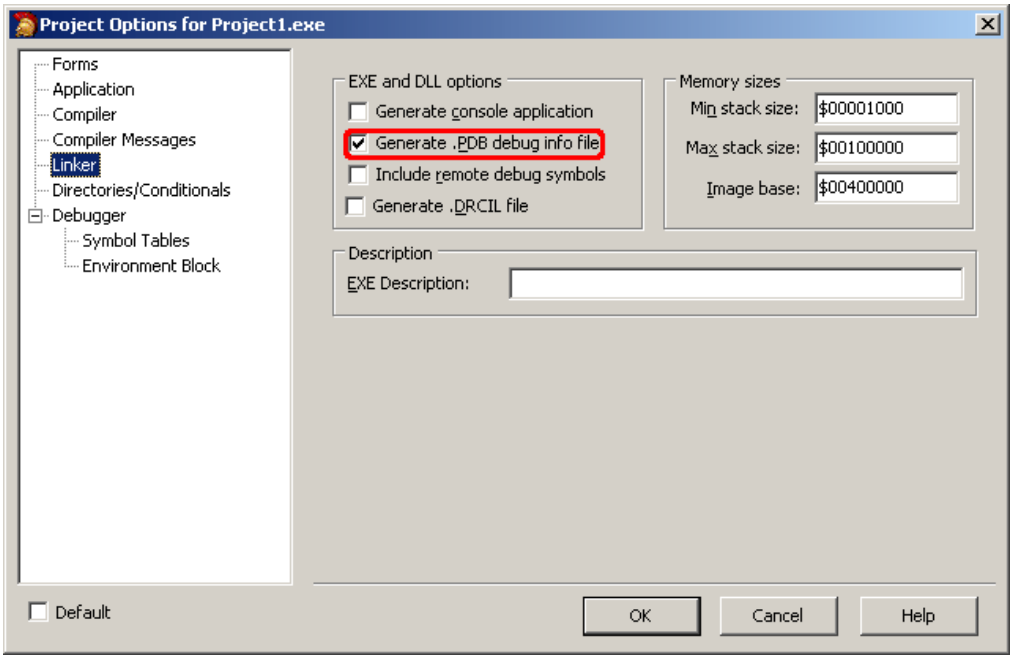
This topic explains how to prepare applications created with Borland Delphi 2006 for .NET for AQtime. To learn how to prepare applications created with other Delphi versions, see *How AQtime Profilers Use Metadata and Debug Information*.

To add debug information to your Delphi 2006 for .NET application, do the following:

1. Open your project in Delphi 2006 for .NET.
2. Select **Project | Options** from the main menu. This will open the **Project Options** dialog.
3. Select the **Compiler** category from the treeview on the left of the dialog and enable the following options:
   - *Debug information*
   - *Local Symbols*
   - *Use Debug DCUILs*



4. Select the **Linker** category and enable the **Generate .PDB debug info file** option:



5. Click **OK** to close the dialog and recompile your application.

When your application is ready for final delivery, do not forget to compile it without debug information to reduce the overall application size.

Compiler Settings for Borland Delphi 2005

The manner, in which you prepare your Borland Delphi 2005 application for AQtime, depends on the kind of your application:

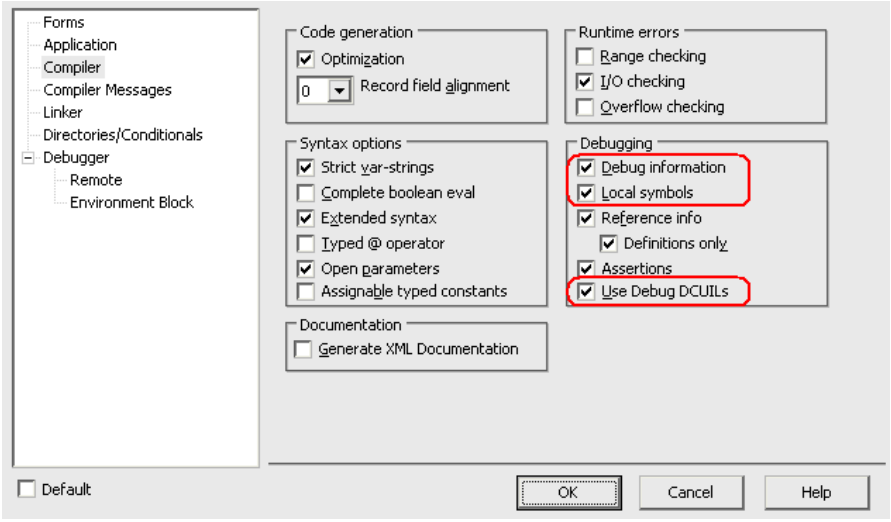
- If you have a *Borland Delphi for Win32* application, use the approach described in *Compiler Settings for Borland Delphi*.
- If you have a *Borland Delphi for .NET* application, use the approach described in *Compiler Settings for Borland Delphi 8 (Delphi for .NET)*.

Compiler Settings for Borland Delphi 8 (Delphi for .NET)

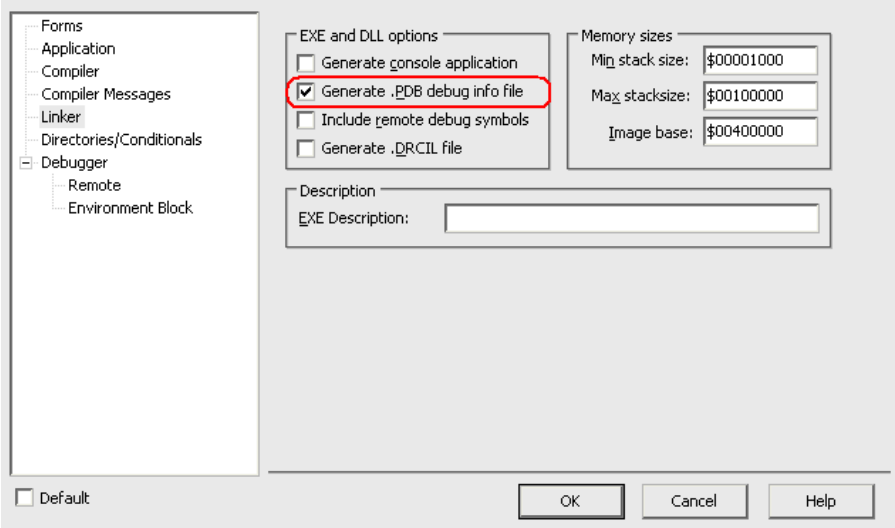
This topic explains how to prepare applications created with Borland Delphi 8 (Delphi for .NET) or Borland Delphi 2005 for .NET for AQtime. To learn how to prepare applications created with Delphi 2005 for Win32 and Delphi 7.0 or earlier, see *Compiler Settings for Borland Delphi*. To learn how to prepare applications created with Delphi 2006 for .NET and Delphi 2006 for Win32, see *Compiler Settings for Borland Delphi 2006 for .NET* and *Compiler Settings for Borland Delphi 2006 for Win32* respectively.

To add debug information to your Delphi for .NET application, do the following:

1. Open your project in Delphi for .NET.
2. Select **Project | Options** from the main menu. This will open the **Project Options** dialog.
3. Select the **Compiler** category from the treeview on the left of the dialog and enable the following options:
   - *Debug information*
   - *Local Symbols*
   - *Use Debug DCUILs*



4. Select the **Linker** category and enable the **Generate .PDB debug info file** option:



5. Click **OK** to close the dialog and recompile your application.

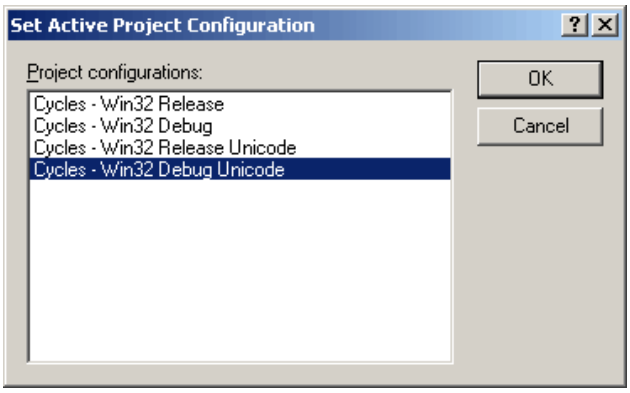
When your application is ready for final delivery, do not forget to compile it without debug information to reduce the overall application size.

Compiler Settings for Microsoft Visual C++ 6.0

This topic explains how to prepare applications that were created with Microsoft Visual C++ 6.0 or earlier for AQtime. To learn how to prepare applications created with Visual C++ 7.x, see *Compiler Settings for Microsoft Visual C++ .NET*. To learn how to prepare applications created with Visual C++ 2005 (8.0), see *Compiler Settings for Microsoft Visual C++ 2005*.

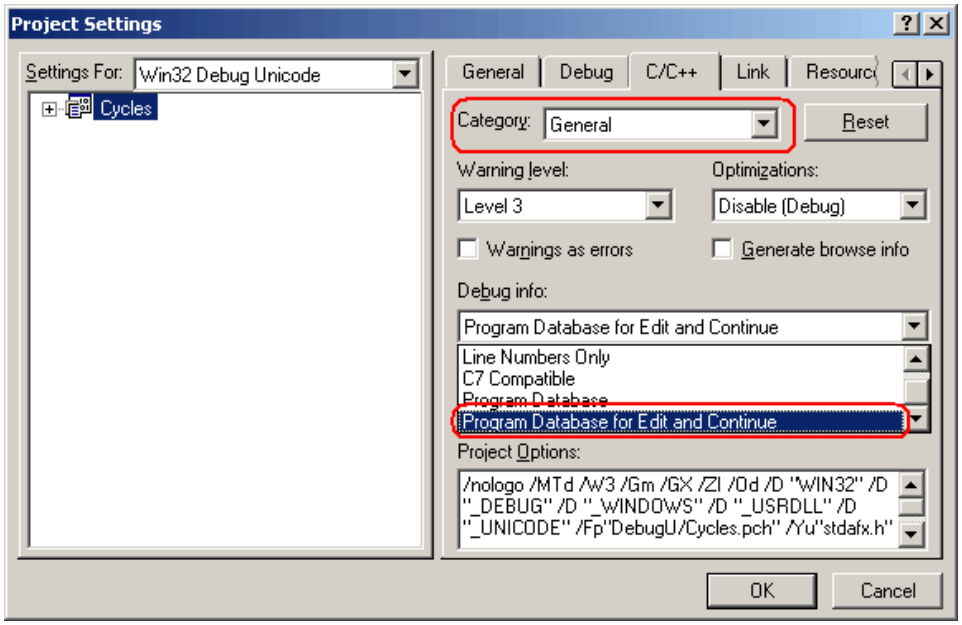
To prepare a Visual C++ application for AQtime, you must ensure that it includes debug info and select the format under which it will be generated. Follow these steps:

1. Open your project in Visual Studio.
2. To prevent the changes from affecting your release-version configuration, do the following: **Choose Build | Set Active Configuration** from Visual C++'s menu and select another configuration, say *Debug*, as the active configuration for the project. All of the changes that are made in the compiler options will be stored to this configuration and will leave the Release configuration unaffected. Usually the Debug configuration is similar to this: *<Your_Project_Name> - Win32 Debug*:



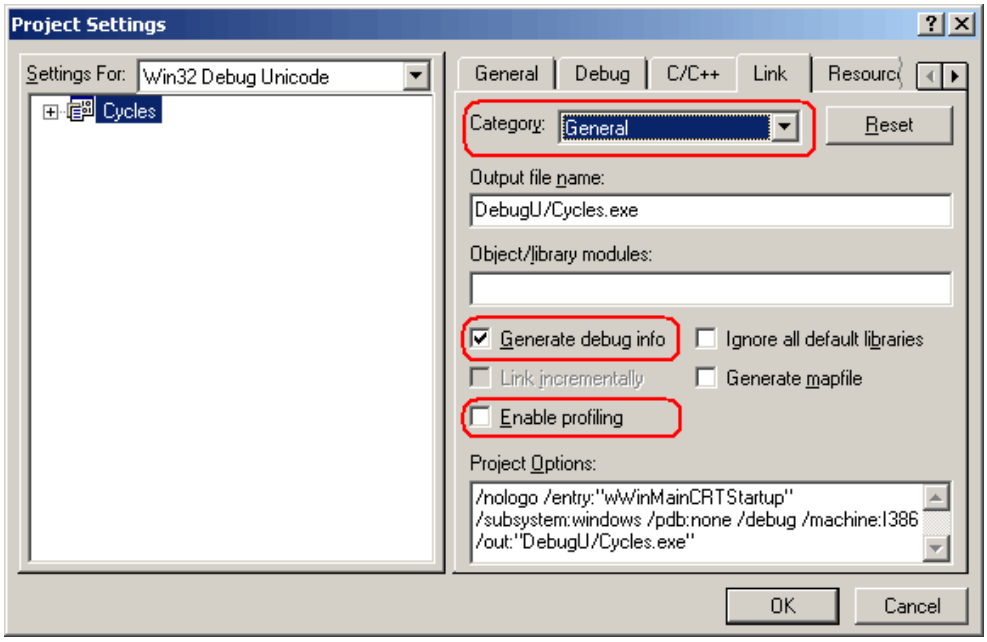
Note that the only reason to change the active configuration is to leave the *Release* configuration unaffected. If you want to profile the *Release* configuration, you may skip this step. In this case, do not forget to restore the compiler settings before compiling the release version of your application.

3. Now, open the **Project Settings** dialog (press *Alt-F7* or use **Project | Settings**) and select the configuration you have set.
4. In the dialog, open the **C/C++** page and make sure that **Debug Info** is set either to "Program Database" or "Program Database for Edit and Continue":



For more information on these options review Microsoft Visual C++ Help.

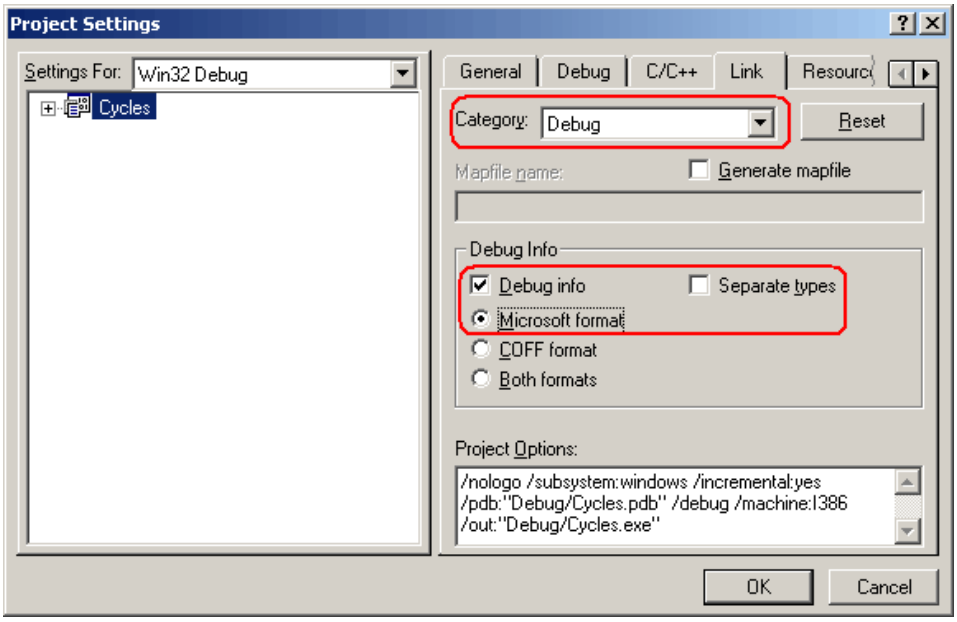
5. You now have set your project to generate debug information when compiling. Next, you must ensure that the linker saves it. From Project | Settings select the **Link** page. There, first set the **Category** to *General*. Then check **Generate debug info** and clear the **Enable profiling check** box:



On the **Link** page, select *Debug* in the **Category** box and then do the following:

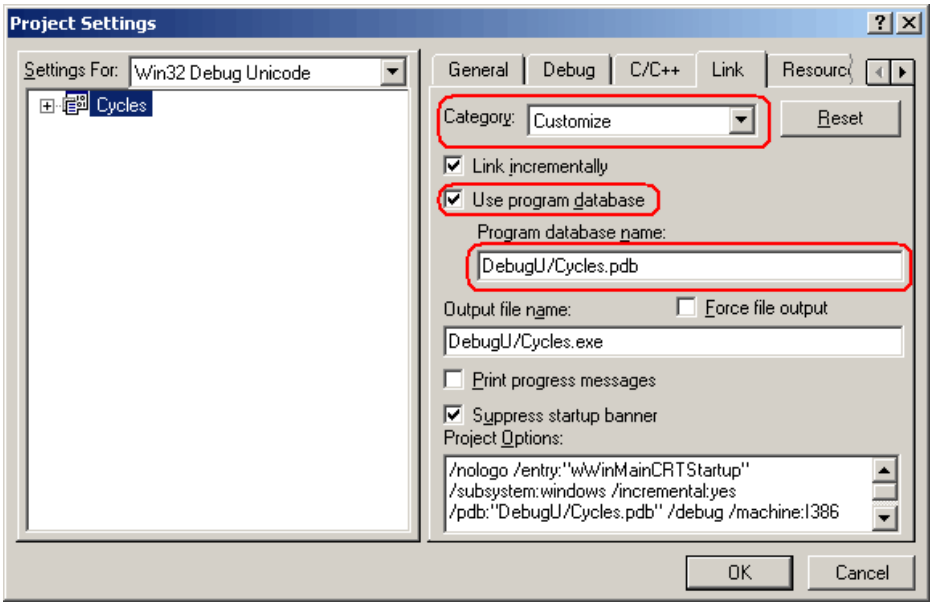
- Select the Debug info check box.

- Clear the Separate type check box.
- Select the Microsoft format option button.



6. The last step is to set how the linker will save the debug information. AQtime supports debug information generated as an external PDB file (PDB format). To set linker options:

- Switch to the **Link** page of the Project Settings dialog:
- Set the **Category** to *Customize*.
- Check **Use program database**.
- Enter the PDB file name you want into the **Program database name** edit field.



Once you have set the compiler and linker options as described above, rebuild your application and it will be ready for profiling. Before opening the application in AQtime, make sure that the output directory of your application includes the .pdb file you have created. In addition, this directory must contain the vcX0.pdb file, where *X* is the major version number of the Visual C++ compiler, e.g. for Visual C++ 6.0 this file is called

vc60.pdb. vcX0.pdb holds part of your application's debugging information, which is needed to profile the entire application in AQtime correctly. If you are profiling an ActiveX control or a COM server, do not forget to register its "debug" version in the system (See *Profiling COM Applications*).

By default, the .pdb debug info file resides in the same folder where the profiled executable or DLL is located. When you open your module in AQtime, it searches for the debug info file in this default lcoation. If you copy the debug info file to another location, then specify this location in the **Symbols Path** list of the Symbols Options dialog (this list contains the search paths). See *Specifying Path to Debug Info Files* for more information.

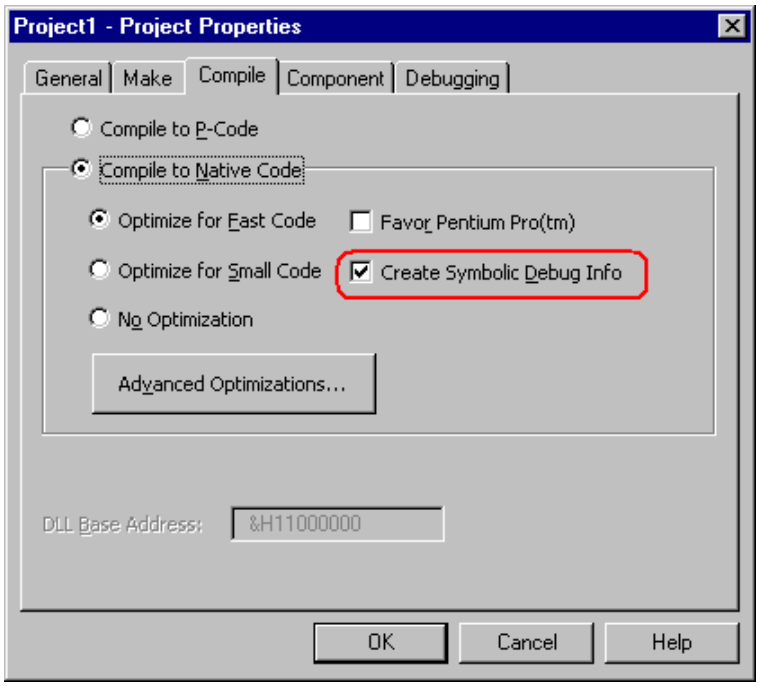
When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Compiler Settings for Microsoft Visual Basic

This topic explains how to prepare applications that were created with Microsoft Visual Basic 6.0 for AQtime. To learn how to prepare applications created with Visual Basic .NET, see *Compiler Settings for Microsoft Visual Basic .NET*. To learn how to prepare applications created with Visual Basic 2005, see *Compiler Settings for Microsoft Visual Basic 2005*.

To prepare your Visual Basic application for AQtime, compile it with debug information that is generated as an external PDB file. Follow these steps:

- Open your project in Microsoft Visual Basic.
- Select **Project | Project Properties** from Visual Basic's main menu. This will open the **Project Properties** dialog.
- Move to the **Compile** tabbed page and check the **Create Symbolic Debug Info** box:



- Press **OK** to close the dialog.
- Before you start compiling your application, make certain that the `Link` environment variable is not defined. If this environment variable exists, Visual Basic will embed debug info into the executable and thus the debug info will be unavailable for AQtime.

If you wish to profile an ActiveX control or a COM server, you must register its "debug" version in the system (see *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Compiler Settings for Embarcadero Delphi 2010 for Win32

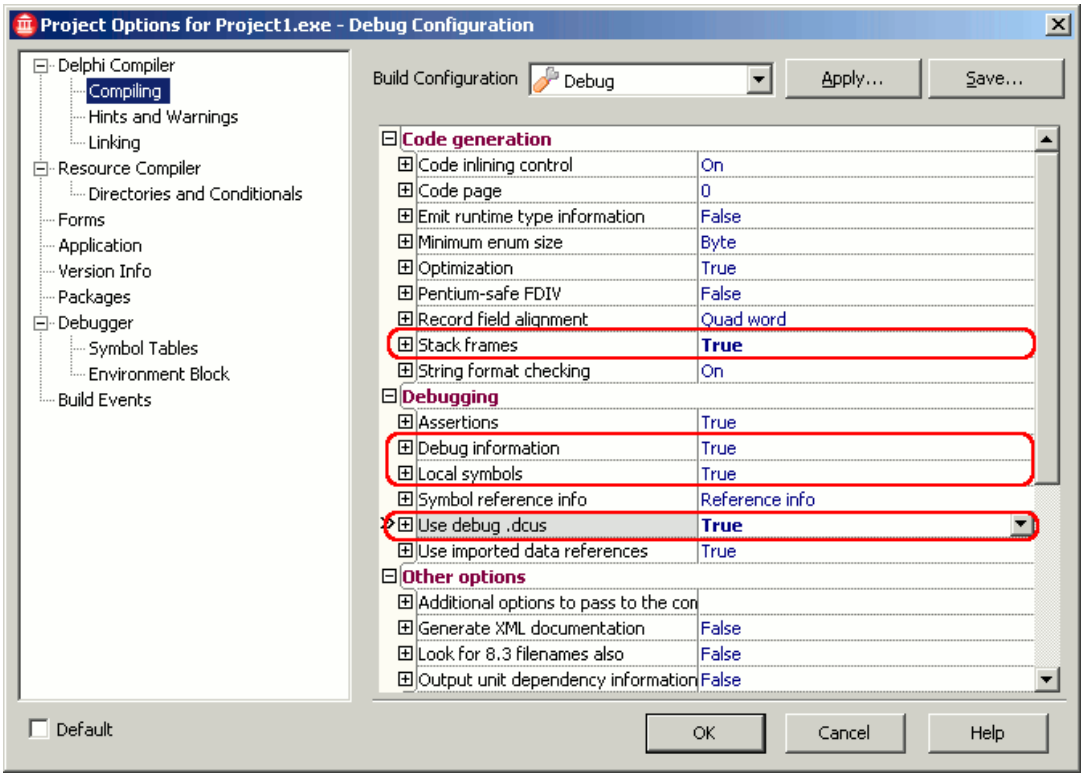
This topic explains how to prepare applications created with Embarcadero Delphi 2010 for Win32 for AQtime. To learn how to prepare applications created with other Delphi versions, see see *How AQtime Profilers Use Metadata and Debug Information*.

To prepare a Delphi 2010 for Win32 application for AQtime, you must first ensure that it includes debug information. Follow these steps:

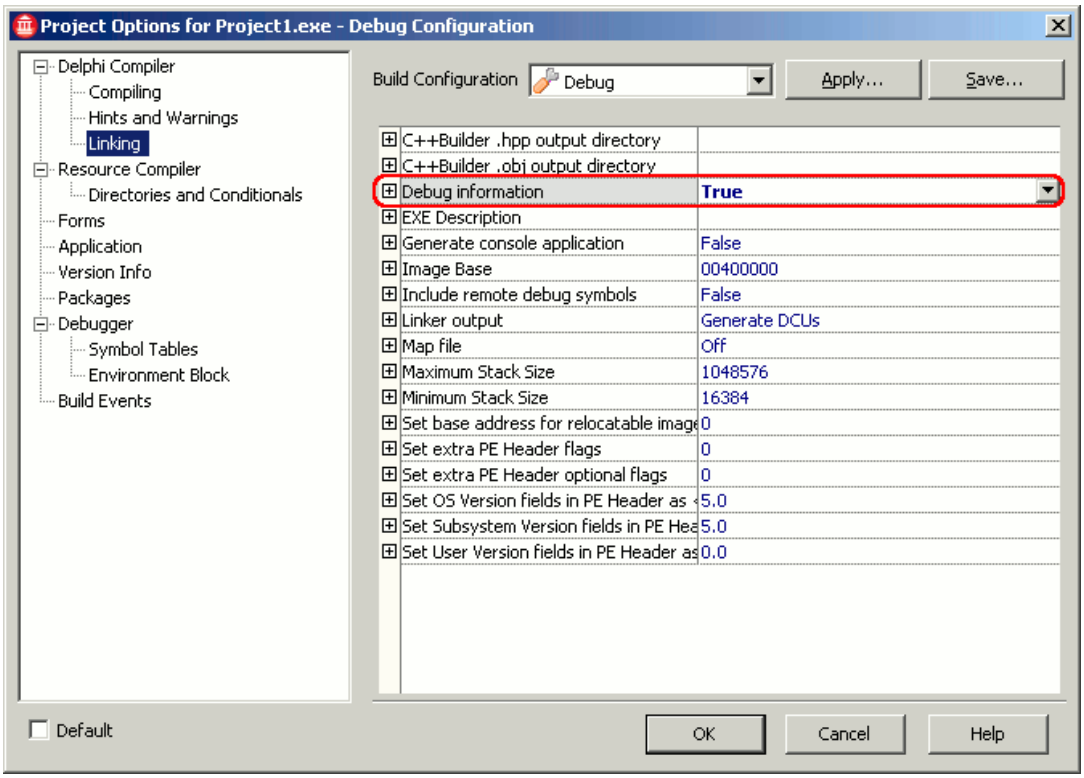
1. Open your Delphi Win32 project in Delphi 2010.
2. Activate the configuration that you use to build the debug version of your application. To do this, right-click the **Project_Name | Build Configurations | Debug Configuration** node in the **Project Manager** and select **Activate** from the context menu.

   Note: You can build your application in any configuration, not just in the debug one. We choose the debug configuration to make sure the changes that will be made to compiler settings will not affect the release configuration that is typically used to build the final version of applications.

3. Choose **Project | Options** from the main menu to open the Project Options dialog.
4. In the **Build Configuration** combo box, select your debug configuration. This will quickly load the settings used for debug builds.
5. To set the compiler options, select the **Delphi Compiler| Compiling** category from the tree view on the left of the dialog.
6. Set the **Stack frames** option to *True* in the **Code generation** group.
7. To include symbolic debug information, set the **Debug information** option to *True* in the **Debugging** group.
8. To view the variables, which are local to procedures and functions, set the **Local symbols** option to *True* in the **Debugging** group.
9. If you want to profile VCL classes, for example, `TDataset`, set the **Use debug .dcus** option to *True* in the **Debugging** group. Otherwise, AQtime will only be able to profile the classes that are defined in your application.

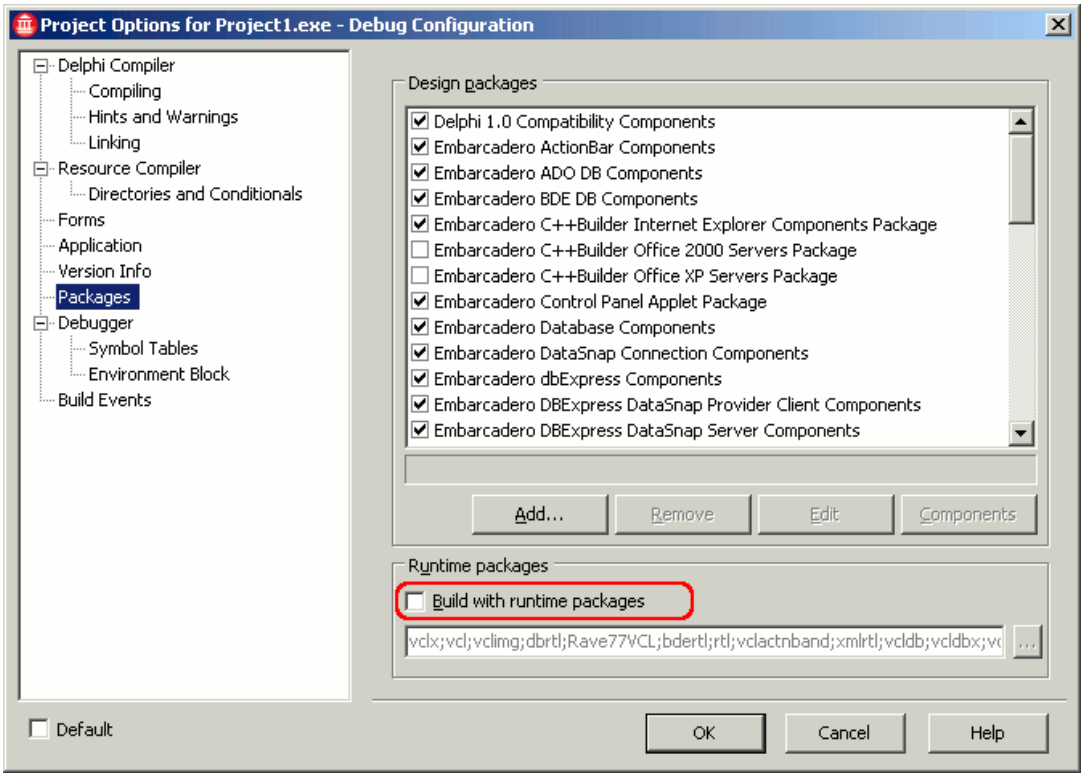


10. Switch to the **Delphi Compiler | Linking** category and set the **Debug information** option to *True*:



11. If you do not want to use the Allocation profiler, skip this step.

Note that the point of the Allocation profiler is not performance. Its point is to track memory allocations and deallocations. To do this, the profiler requires access to the basic VCL objects (`TObject` and `TInterfacedObject`). Therefore, the easiest way to provide this access is to turn off the **Build with runtime packages** option in the **Packages** category:



Note: If you want to keep *Build with runtime packages* (for instance, to control the exe size), you can still use the Allocation profiler. When you include your application in an AQtime project, you will also have to include the <Windows>\System32\RTL100.BPL file.

To add a module to an AQtime project, press **Add Module** on the Setup toolbar or select it from the Setup context menu.

12. Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its "debug" version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Note: AQtime is incompatible with some third-party tools that modify the binary code of your application (for example, those that add a custom exception handling mechanism). An example of such a tool is EurekaLog. We recommend that you profile your application before processing it with such tools.

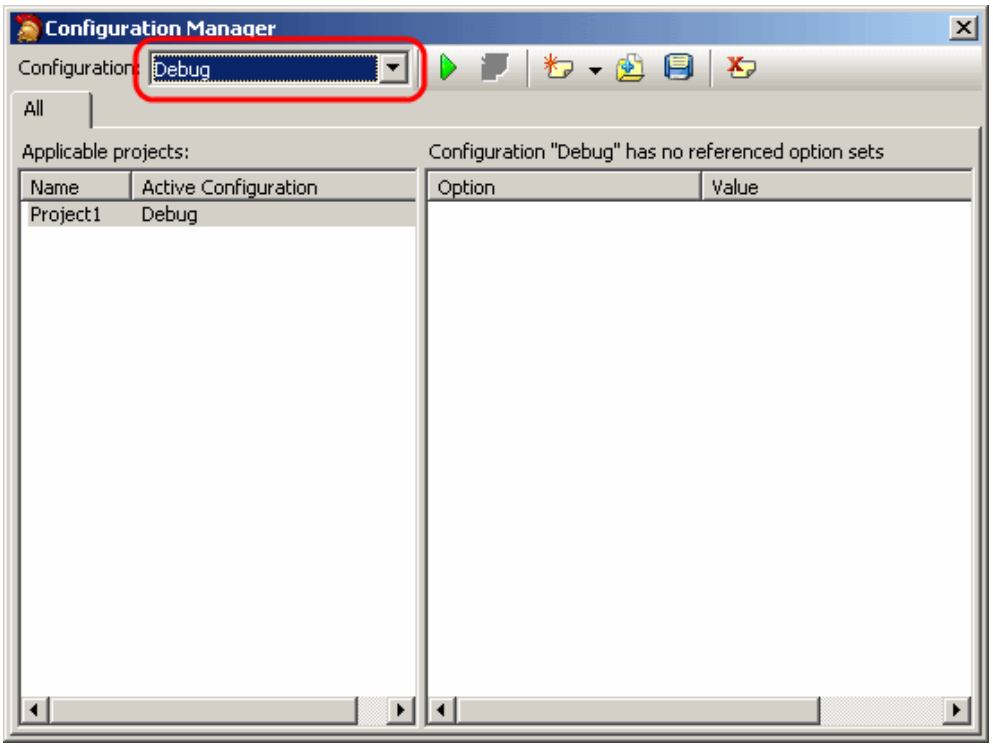
Nevertheless, AQtime is compatible with AQtrace and supports profiling of applications that use AQtrace for error reporting.

Compiler Settings for CodeGear Delphi 2009 for Win32

This topic explains how to prepare applications created with CodeGear Delphi 2009 for Win32 for AQtime. To learn how to prepare applications created with other Delphi versions, see *How AQtime Profilers Use Metadata and Debug Information*.

To prepare a Delphi 2009 for Win32 application for AQtime, you must first ensure that it includes debug information. Follow these steps:

1. Open your project in CodeGear Delphi 2009 for Win32.
2. Select **Project | Configuration Manager** from the main menu. This will open the **Build Configuration Manager** dialog. Select the *Debug* configuration for your project:

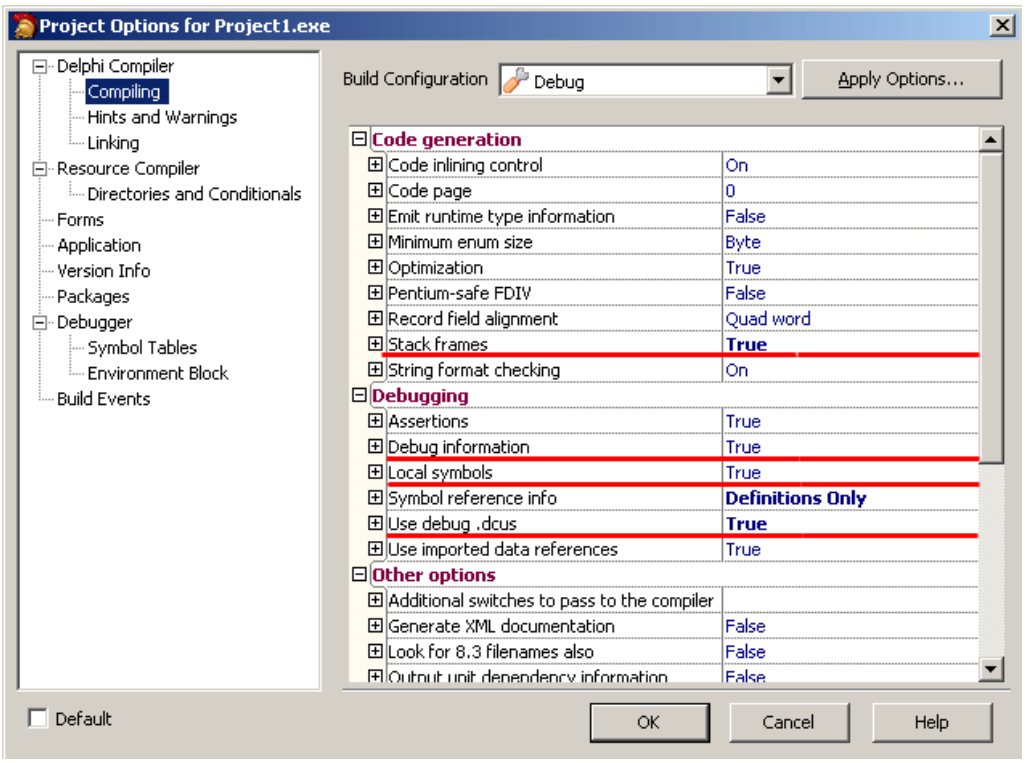


   Close the dialog.

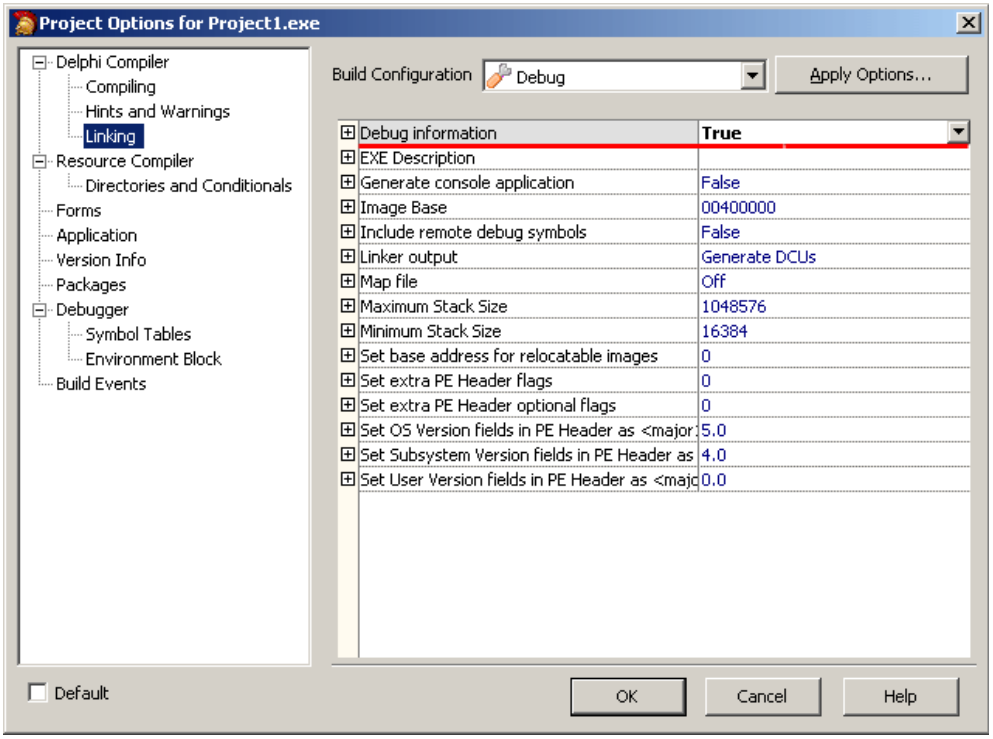
Note: You can profile your application in any configuration, not just in the *Debug* one. We chose the Debug configuration to make sure the changes that are made to compiler settings will not affect the *Release* configuration, which is typically used to build the final version of applications.

3. Select **Project | Options** from the main menu. This will open the **Project Options** dialog.
4. To set the compiler options, select the **Delphi Compiler| Compiling** category from the treeview on the left of the dialog.
5. To include the symbolic debug information, set the **Debug information** option to **True** in the **Debugging** section.
6. To view the variables local to procedures and functions, set the **Local symbols** option to **True**.
7. If you want to profile VCL classes, for example, TDataset, set the Use **Debug .DCUs** option to **True**. Otherwise, AQtime will only be able to profile the classes that are defined in your application.

© 2010 AutomatedQA Corp.

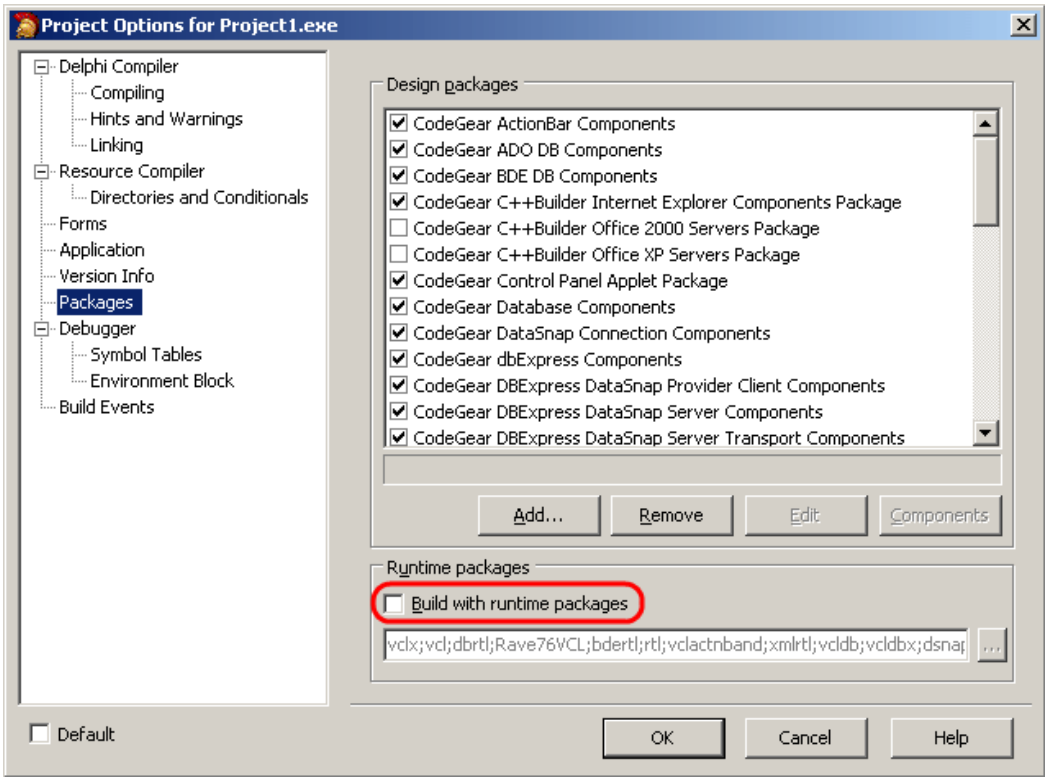


8. To set the linker options, select the **Delphi Compiler | Linking** category from the treeview on the left of the dialog.
9. Set the **Debug information** option to **True**:



10. If you do not want to use the *Allocation* profiler, skip this step.

11. Note that the point of the Allocation profiler is not performance. Its point is to track memory allocations and deallocations. To do this, the profiler requires access to the basic VCL objects (TObject and TInterfacedObject). Therefore, the easiest way to provide this access is to uncheck the Build with runtime packages box in the **Resource Compiler | Packages** category:



Note: If you want to keep *Build with runtime packages* (for instance, to control the exe size), you can still use the Allocation profiler. When you include your application in an AQtime project, you will also have to include the <Windows>\System32\RTL100.BPL file.
To add a module to an AQtime project, press **Add Module** on the Setup toolbar or select it from the Setup context menu.

12. Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its "debug" version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Note: AQtime is incompatible with some third-party tools that modify the binary code of your application (for example, those that add a custom exception handling mechanism). An example of such a tool is EurekaLog. We recommend that you profile your application before processing it with such tools.

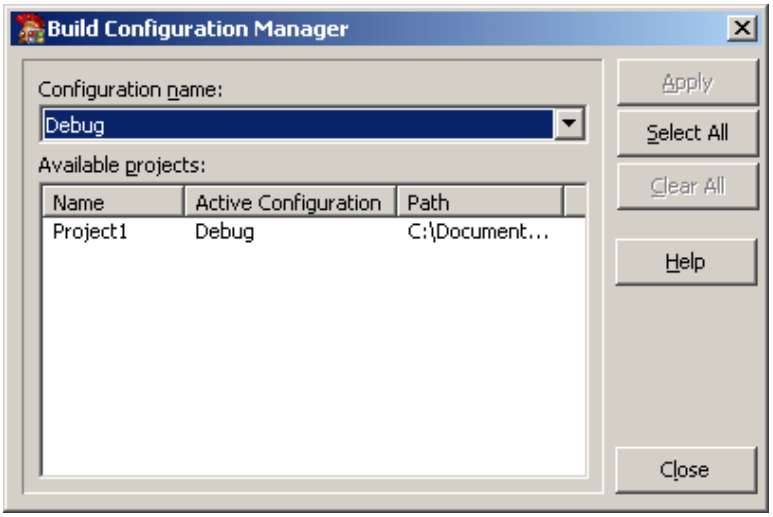
Nevertheless, AQtime is compatible with AQtrace and supports profiling of applications that use AQtrace for error reporting.

Compiler Settings for CodeGear Delphi 2007 for Win32

This topic explains how to prepare applications created with CodeGear Delphi 2007 for Win32 for AQtime. To learn how to prepare applications created with other Delphi versions, see *How AQtime Profilers Use Metadata and Debug Information*.

To prepare a CodeGear Delphi 2007 for Win32 application for AQtime, you must first ensure that it includes the TD32 debug info. Follow these steps:

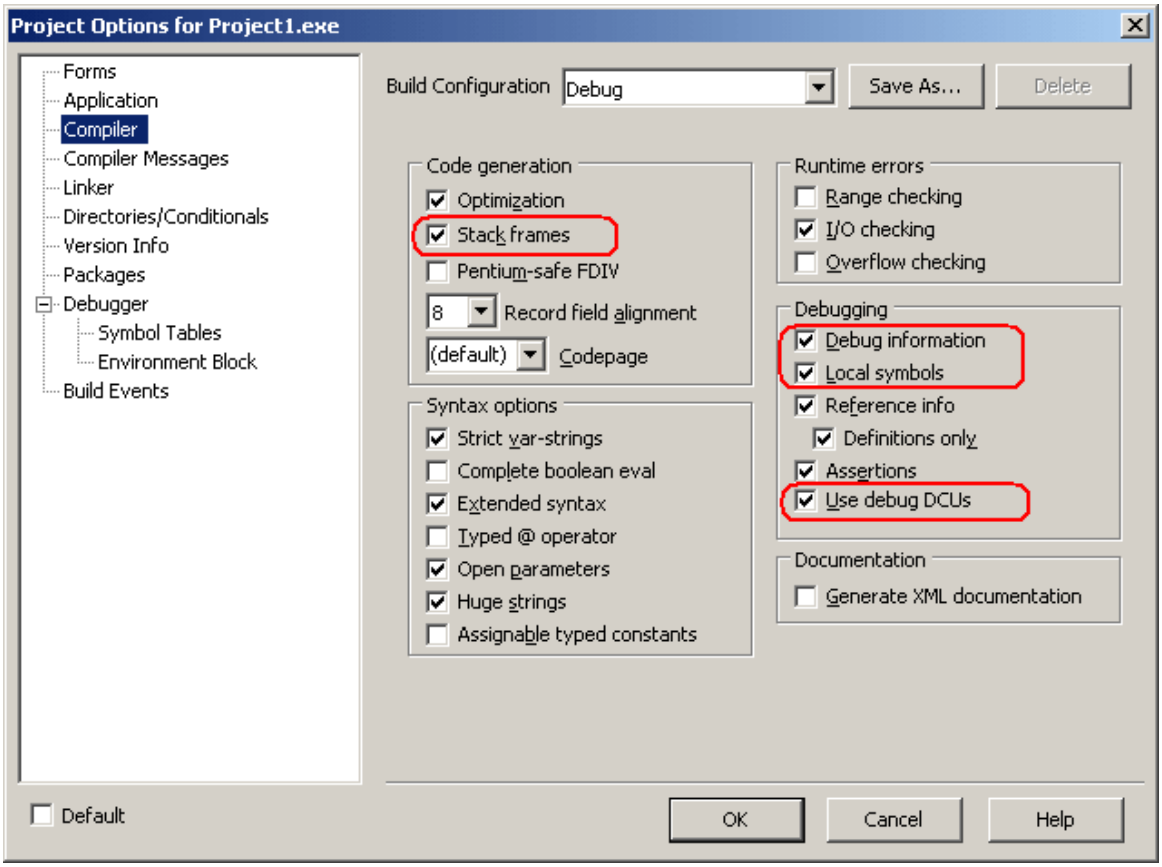
1. Open your project in CodeGear Delphi 2007 for Win32.
2. Select **Project | Configuration Manager** from the main menu. This will open the **Build Configuration Manager** dialog. Select the *Debug* configuration for your project:



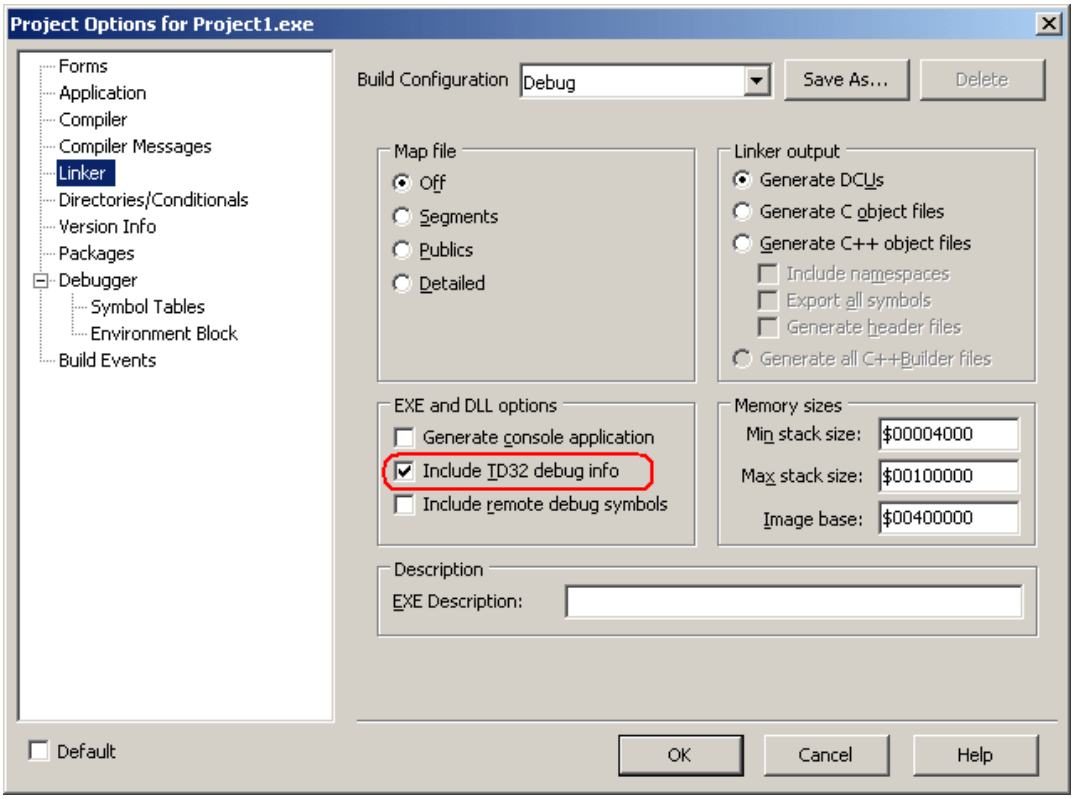
   Close the dialog.

   Note: You can profile your application in any configuration, not just in the *Debug* one. We chose the Debug configuration to make sure the changes that will be made to compiler settings will not affect the *Release* that configuration, which is typically used to build the final version of applications.

3. Select **Project | Options** from the main menu. This will open the **Project Options** dialog.
4. To set the compiler options, select the **Compiler** category from the treeview on the left of the dialog.
5. To include the symbolic debug information, in the **Debugging** section of the Compiler page, check **Debug information**.
6. To view the variables local to procedures and functions, check **Local symbols**.
7. If you want to profile VCL classes, for example, `TDataset`, check the **Use Debug DCUs** box. Otherwise, AQtime will be able to profile only the classes that are defined in your application.

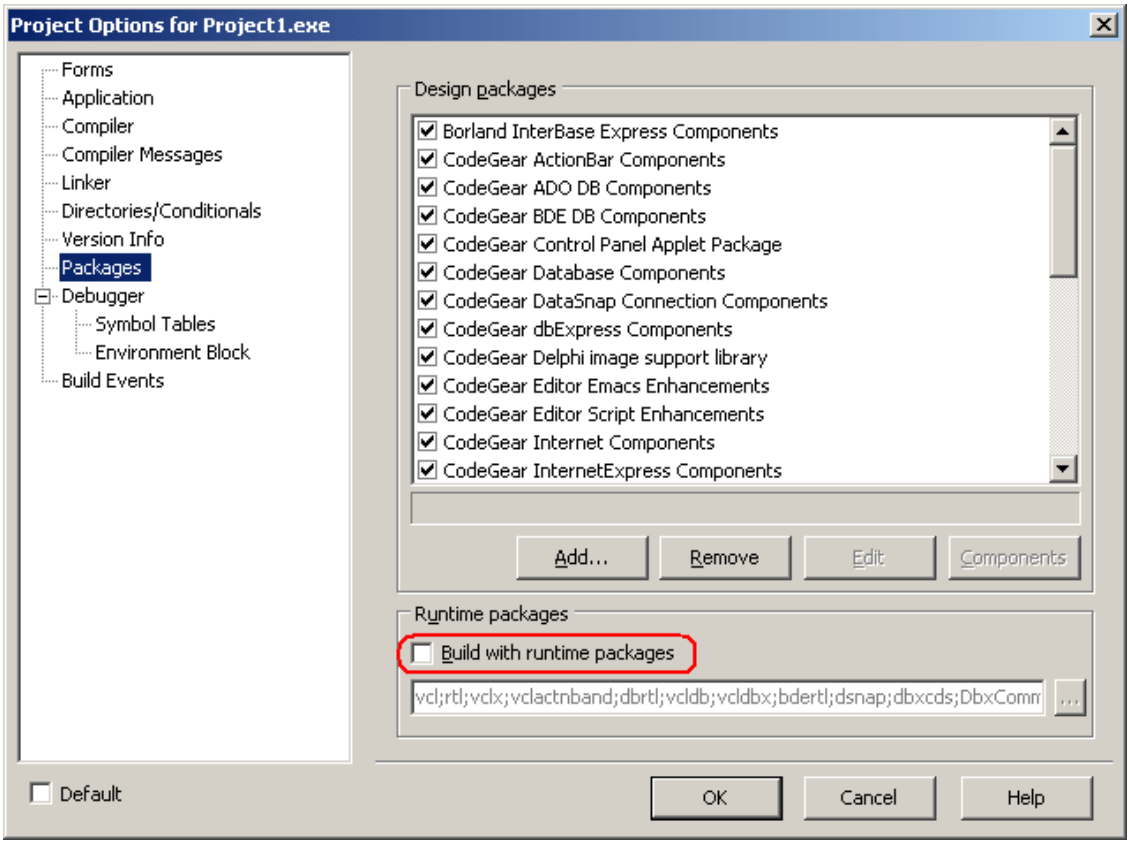


8. To set the linker options, select the **Linker** category from the treeview on the left of the dialog.
9. In the **EXE and DLL options** group, check **Include TD32 debug info**:



10. Now, if you do not want to use the Allocation profiler, go to point 11.

Note that the point of the Allocation profiler is not performance. Its point is to track memory allocations and deallocations. To do this, the profiler requires access to the basic VCL objects (`TObject` and `TInterfacedObject`). Therefore, the easiest way to provide this access is to uncheck the **Build with runtime packages** box in the **Packages** category:



Note: If you want to keep *Build with runtime packages* (for instance, to control the exe size), you can still use the Allocation profiler. When you include your application in an AQtime project, you will also have to include the <Windows>\System32\RTL100.BPL file.
To add a module to an AQtime project, press **Add Module** on the Setup toolbar or select it from the Setup context menu.

11. Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its "debug" version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Note: AQtime is incompatible with some third-party tools that modify the binary code of your application (for example, those that add a custom exception handling mechanism). An example of such a tool is EurekaLog. We recommend that you profile your application before processing it with such tools.

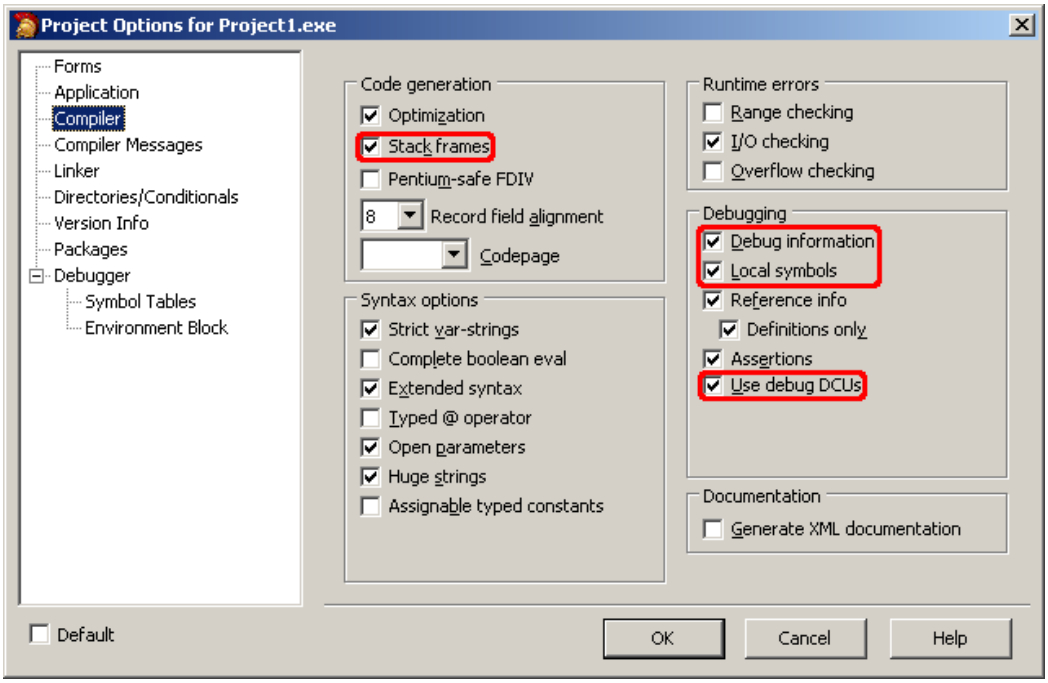
Nevertheless, AQtime is compatible with AQtrace and supports profiling of applications that use AQtrace for error reporting.

Compiler Settings for Borland Delphi 2006 for Win32

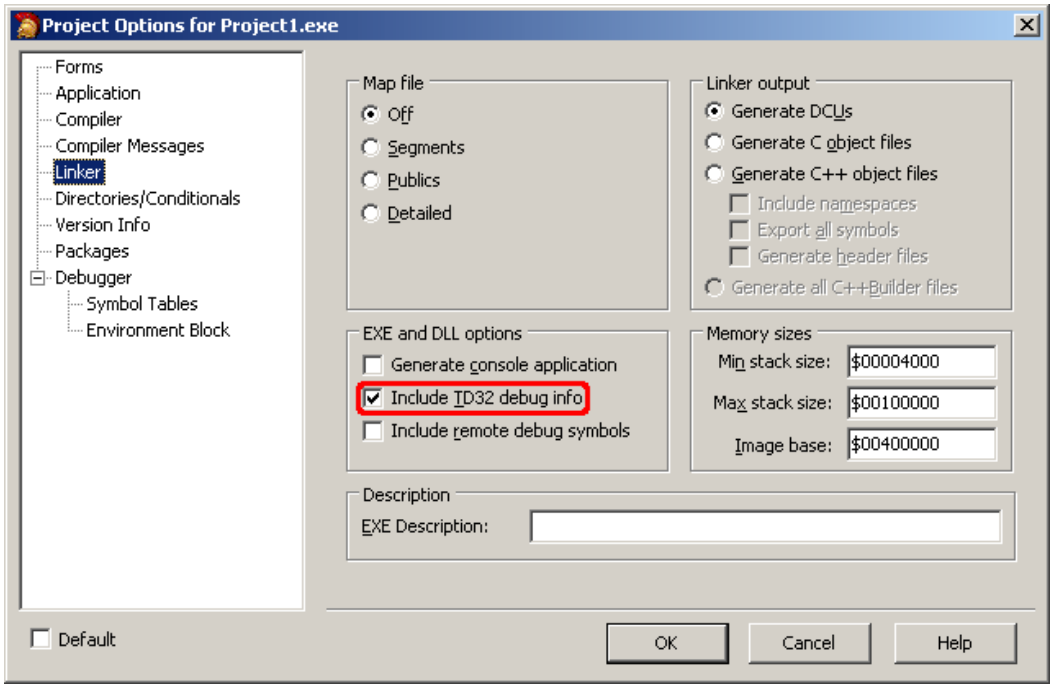
This topic explains how to prepare applications created with Borland Delphi 2006 for Win32 for AQtime. To learn how to prepare applications created with Delphi 2006 for .NET, see *Compiler Settings for Borland Delphi 2006 for .NET*.

To prepare a Delphi 2006 for Win32 application for AQtime, you must first ensure that it includes the TD32 debug info. Follow these steps:

1. Open your project in Delphi 2006 for Win32.
2. Select **Project | Options** from the main menu. This will open the **Project Options** dialog.
3. To set the compiler options, select the **Compiler** category from the treeview on the left of the dialog.
4. To include the symbolic debug information, in the **Debugging** section of the Compiler page, check **Debug information**.
5. To view the variables local to procedures and functions, check **Local symbols**.
6. If you want to profile VCL classes, for example, `TDataset`, check the **Use Debug DCUs** box. Otherwise, AQtime will be able to profile only the classes that are defined in your application.

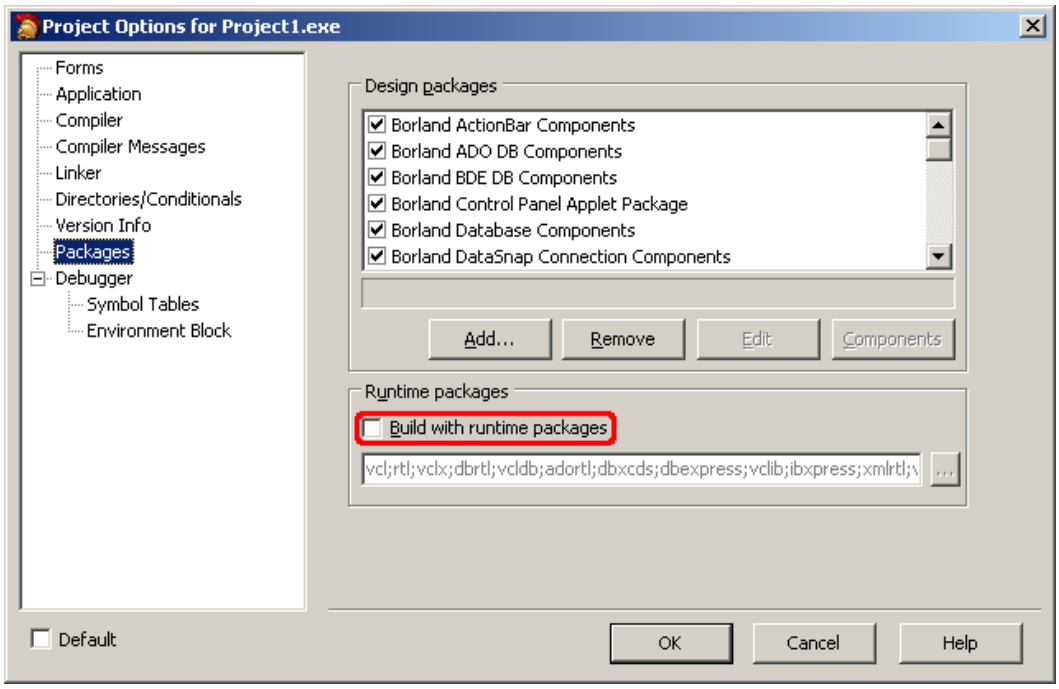


7. To set the linker options, select the **Linker** category from the treeview on the left of the dialog.
8. In the **EXE and DLL options** group, check **Include TD32 debug info**:



9. Now, if you do not want to use the Allocation profiler, go to point 10.

Note that the point of the Allocation profiler is not performance. Its point is to track memory allocations and deallocations. To do this, the profiler requires access to the basic VCL objects (`TObject` and `TInterfacedObject`). Therefore, the easiest way to provide this access is to uncheck the **Build with runtime packages** box in the **Packages** category:



If you want to keep Build with runtime packages (for instance, to control the exe size), you can still use the Allocation profiler. When you include your application in an AQtime project, you will also have to include the

<Windows>\System32\RTL100.BPL file. To add a module to an AQtime project, press **Add Module** on the Setup toolbar or select it from the Setup context menu.

10. Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its "debug" version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Note: AQtime is incompatible with some third-party tools that modify the binary code of your application (for example, those that add a custom exception handling mechanism). An example of such a tool is EurekaLog. We recommend that you profile your application before processing it with such tools.

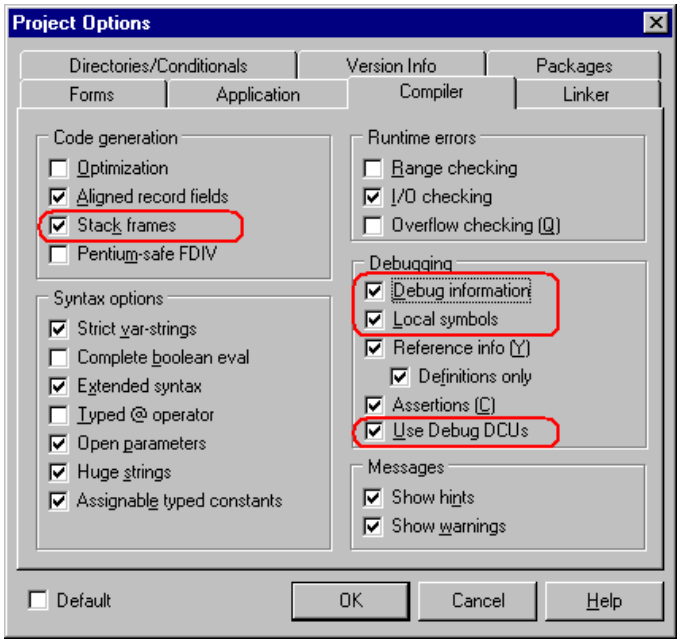
Nevertheless, AQtime is compatible with AQtrace and supports profiling of applications that use AQtrace for error reporting.

Compiler Settings for Borland Delphi 3-7

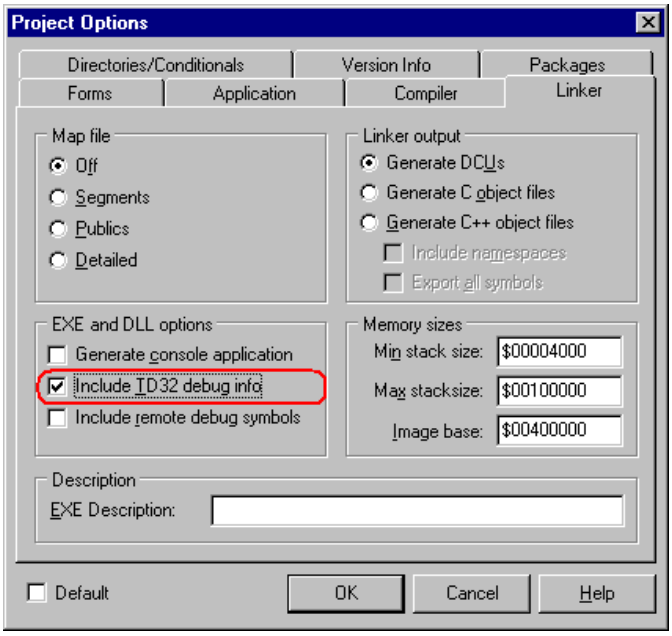
This topic explains how to prepare applications that were created with Borland Delphi 3.0 - 7.0. The described steps are also applicable to Borland Delphi 2005 for Win32 projects. To learn how to prepare applications created with other Delphi versions, see *How AQtime Profilers Use Metadata and Debug Information*.

To prepare a Delphi application for AQtime, you must first ensure that it includes the TD32 debug info. Follow these steps:

1. Open your project in Borland Delphi.
2. To set the compiler options, choose **Project | Options** from Delphi's main menu and select the **Compiler** tabbed page.
3. To include the symbolic debug information, in the **Debugging** section of the Compiler page, check **Debug information**.
4. To view the variables local to procedures and functions, check **Local Symbols**.
5. If you want to profile VCL classes, for example, `TDataset`, check the **Use Debug DCUs** box (it is available in Delphi 5 or later). Otherwise, AQtime will be able to profile only the classes that are defined in your application.

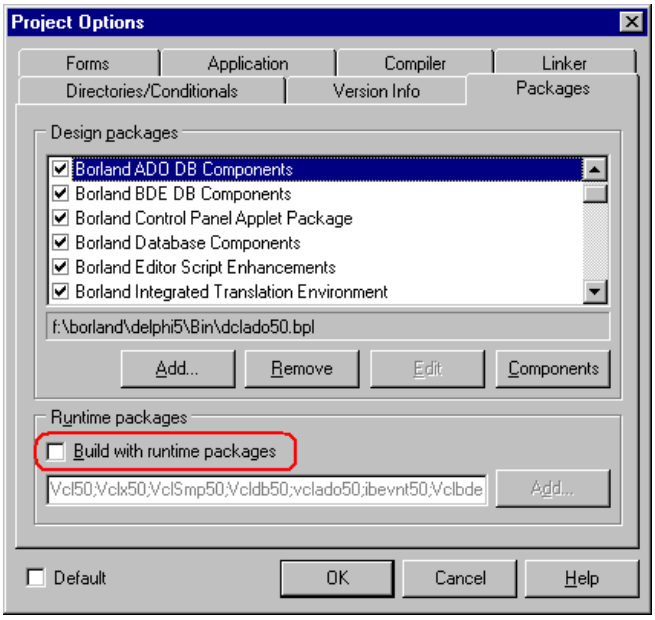


6. To set the linker options, select the **Linker** tabbed page. In the **EXE and DLL options** group, check **Include TD32 debug info**:



7. Now, if you do not want to use the Allocation profiler, go to point 8.

   Note that the point of the Allocation profiler is not performance. Its point is to track memory allocations and deallocations. To do this, the profiler requires access to the basic VCL objects (`TObject` and `TInterfacedObject`). Therefore, the easiest way to provide this access is to uncheck the **Build with runtime packages** box on the **Packages** tabbed page:



If you want to keep Build with runtime packages (for instance, to control the exe size), you can still use the Allocation profiler. When you include your application in an AQtime project, you will also have to include the <Windows>\System32\VCL*nn*.BPL or <Windows>\System32\RTL*nn*.BPL file, where *nn* is the compiler main version number, followed by 0. For instance:

- Add the VCL50.BPL file if you use Delphi 5.
- Add the RTL60.BPL file if you use Delphi 6.
- Add the RTL70.BPL file if you use Delphi 7.
- Add the RTL90.BPL file if you use Delphi 2005 for Win32.

To add a module to an AQtime project, press **Add Module** on the Setup toolbar or select it from the Setup context menu.

8. Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its “debug” version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Note: AQtime is incompatible with some third-party tools that modify the binary code of your application (for example, those that add a custom exception handling mechanism). An example of such a tool is EurekaLog. We recommend that you profile your application before processing it with such tools.

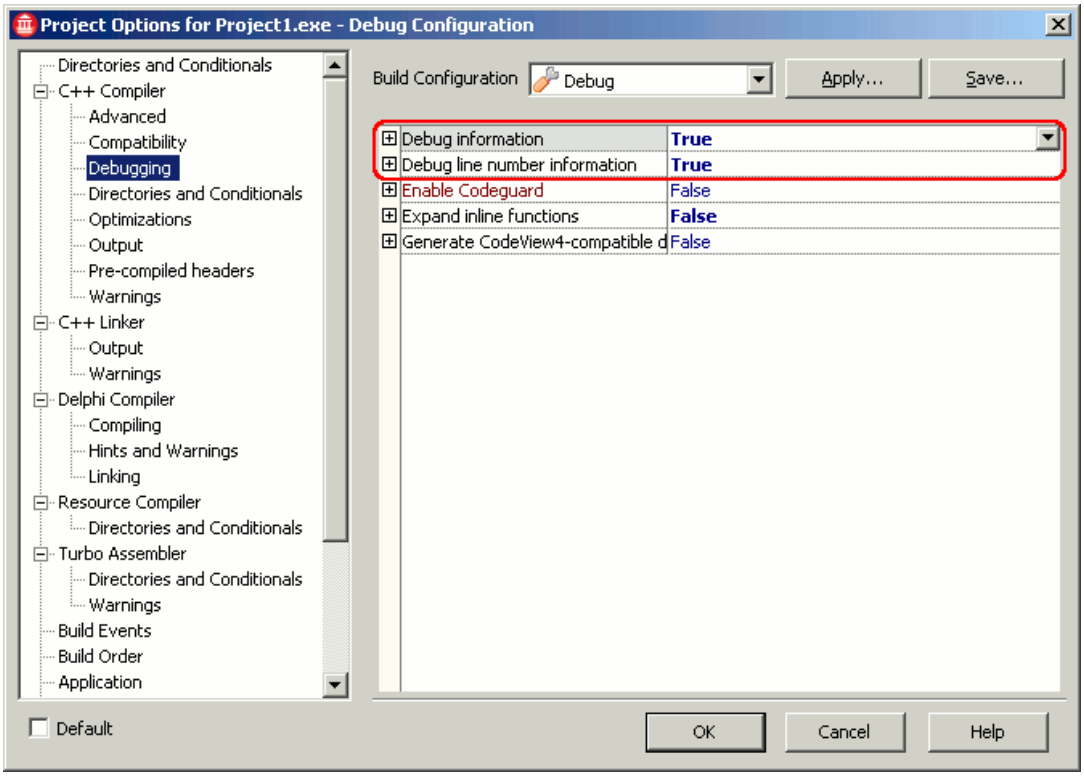
Nevertheless, AQtime is compatible with AQtrace and supports profiling of applications that use AQtrace for error reporting.

Compiler Settings for Embarcadero C++Builder 2010

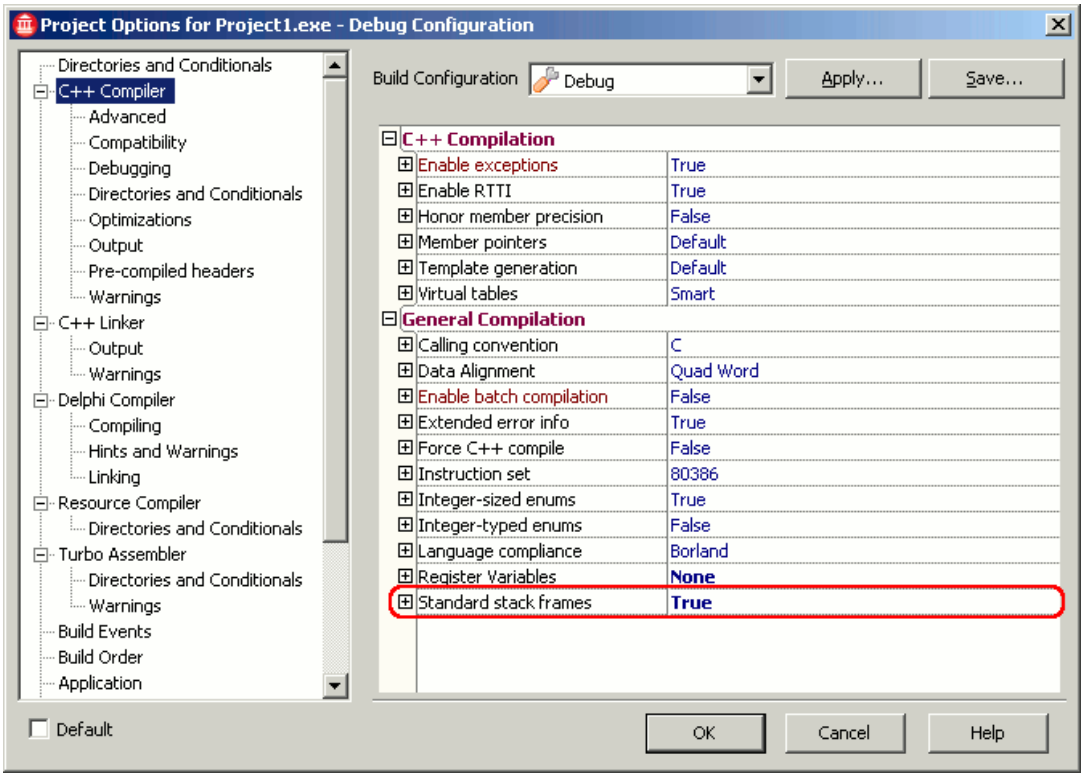
This topic explains how to prepare applications created with Embarcadero C++Builder 2010 for AQtime. To learn how to prepare applications created with other C++Builder versions, see *How AQtime Profilers Use Metadata and Debug Information*.

To prepare an Embarcadero C++Builder 2010 application for AQtime, you must first ensure that it includes debug information. Follow these steps:

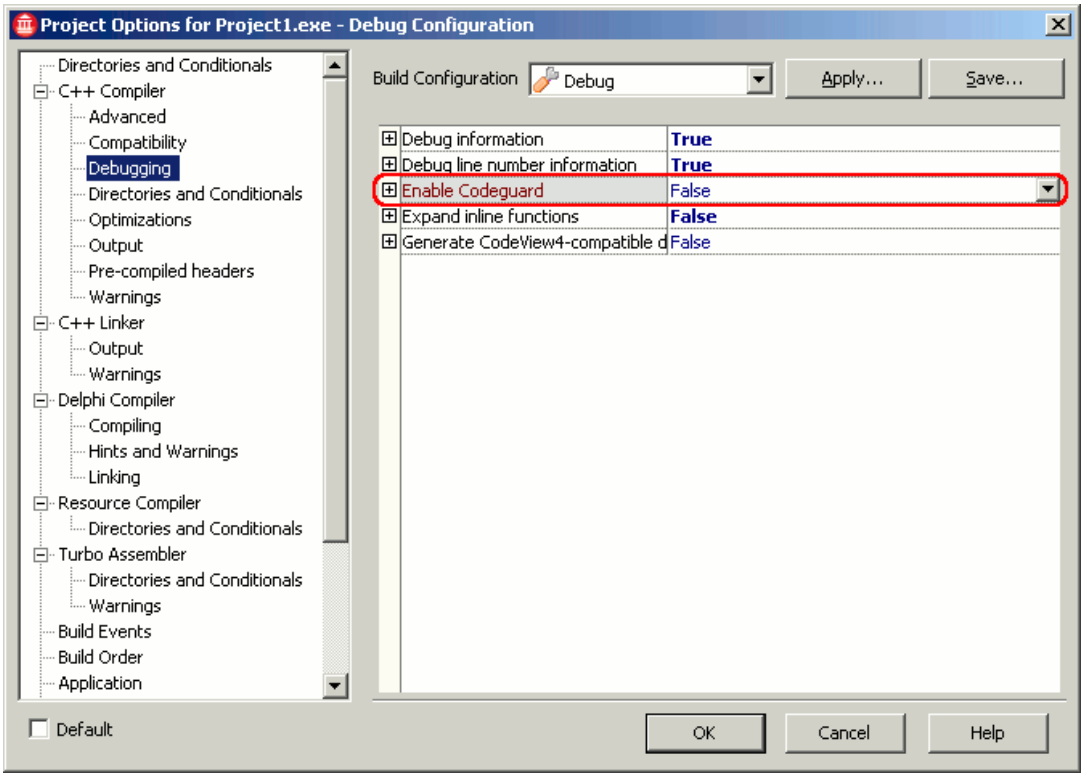
1. Open your project in C++Builder 2010.
2. Choose **Project | Options** from the main menu to open the Project Options dialog.
3. To set the C++ compiler options, select the **C++ Compiler | Debugging** category from the tree view on the left of the dialog.
4. To include symbolic debug information, set the **Debug information** option to *True*. In addition, to refer this information to source line numbers, set the **Debug line number information** option to *True*.



5. (Optional) To generate stack frames when using the C++ compiler, switch to the **C++ Compiler** category and set the **Standard stack frames** option to *True*.

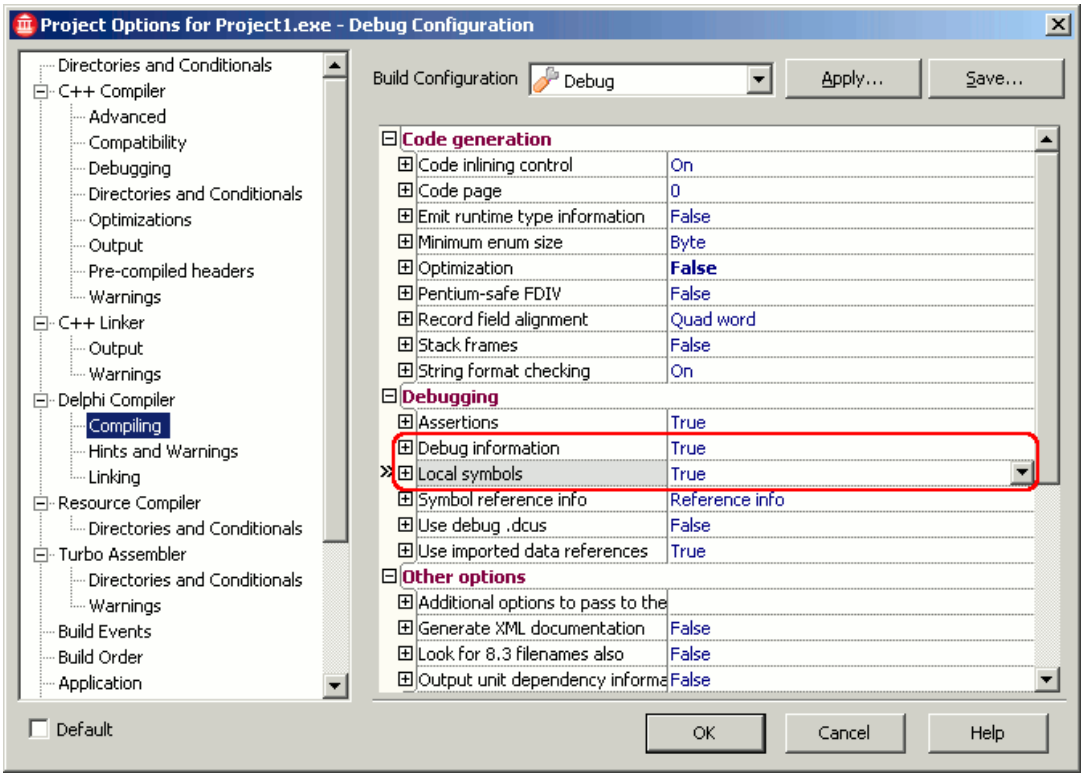


6. If you compile your application with CodeGuard enabled, the latter will report errors during the profiler execution. They relate to AQtime hooking code and are not actually errors. We recommend that you disable CodeGuard before profiling your application with AQtime. To do this, switch to the **C++ Compiler | Debugging** category and set the **Enable CodeGuard** option to *False*.

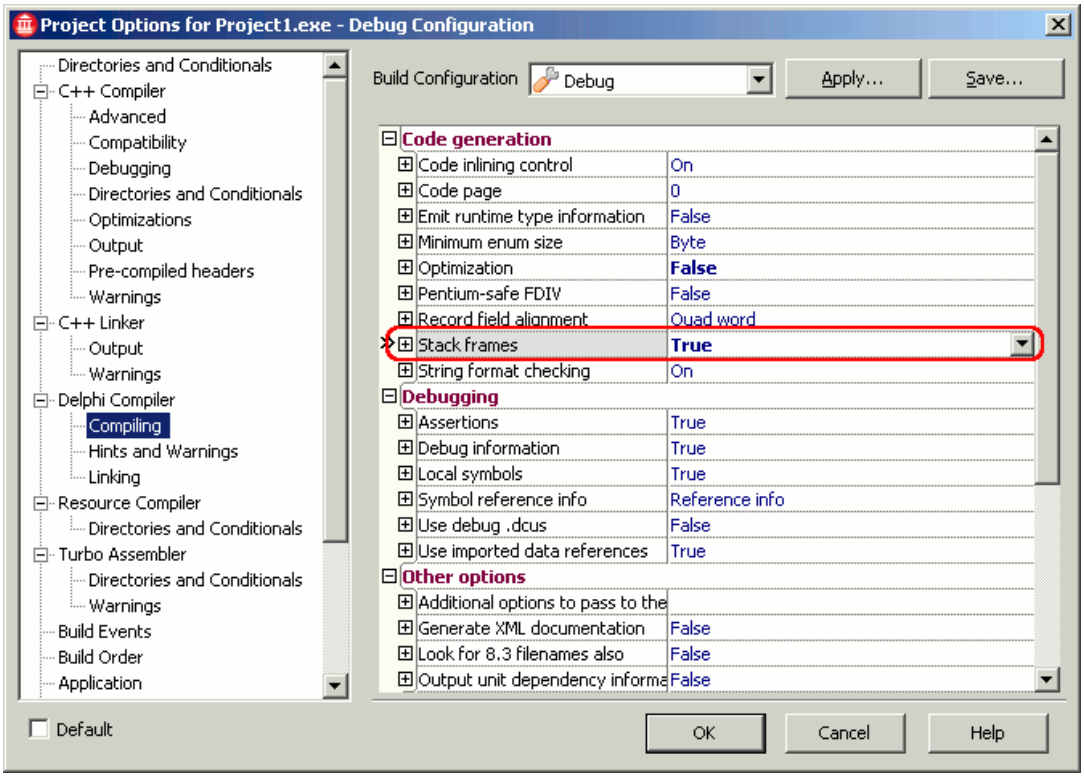


7. To set the Delphi compiler options, select the **Delphi Compiler | Compiling** category from the tree view on the left of the dialog.

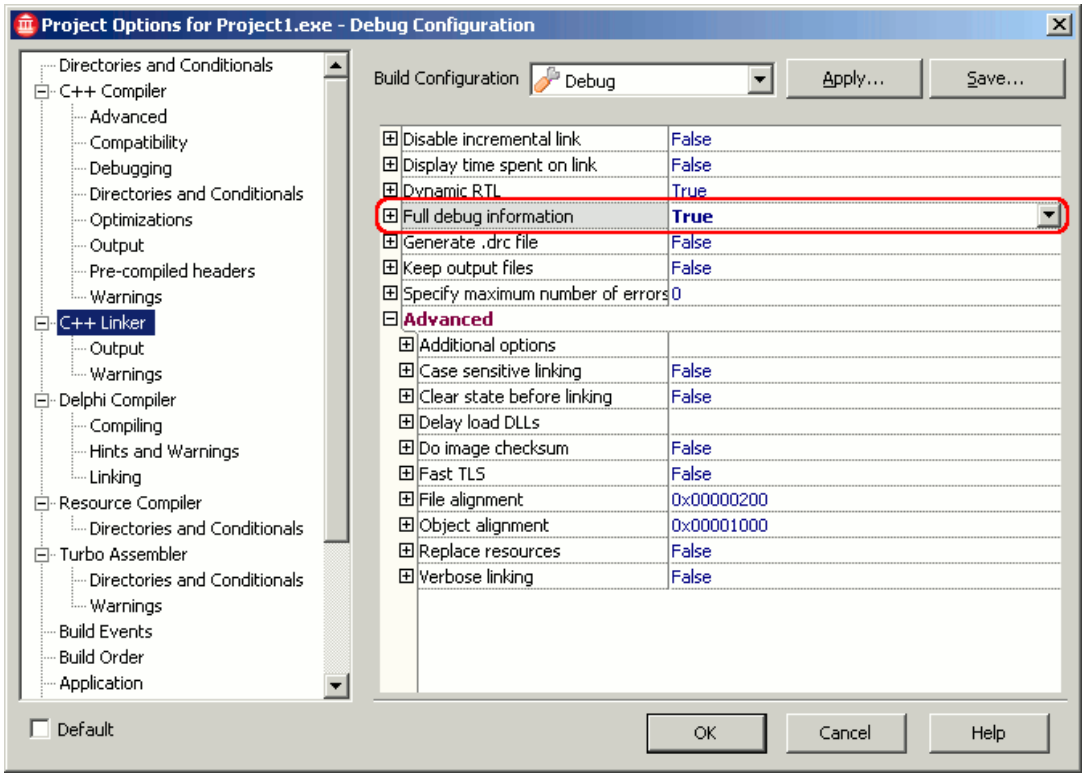
8. To include symbolic debug information, set the **Debug information** option to *True*. In addition, to refer this information to source line numbers, set the **Local symbols** option to *True*.



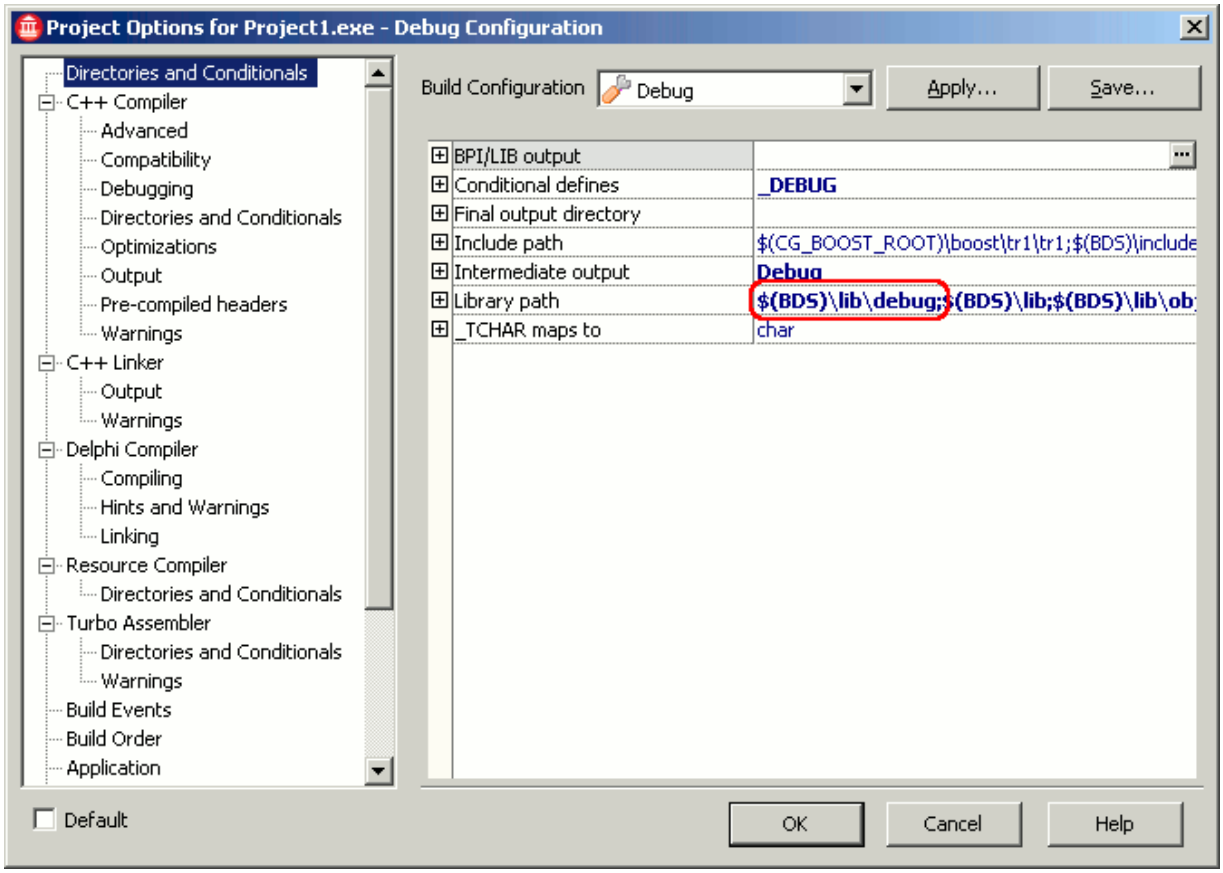
9. (Optional) To generate stack frames when using the Delphi compiler, switch to the **Delphi Compiler | Compiling** category and set the **Stack frames** option to *True*.



10. Switch to the **C++ Linker** category and set the **Full debug information** option to *True*:

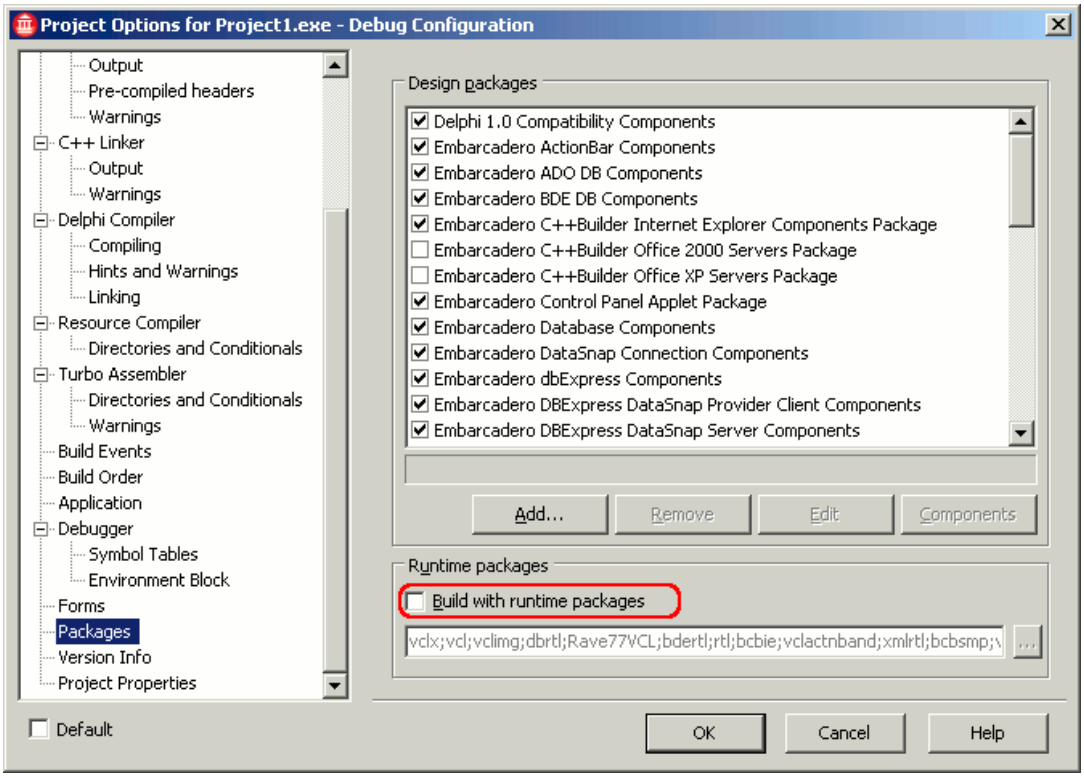


11. Switch to the **Directories and Conditionals** category and examine the **Library path** option. If the path contains the *$(BDS)\lib\release* folder, replace it with *$(BDS)\lib\debug*:



12. Now, if you do not want to use the Allocation profiler, skip this step.

The point of the Allocation profiler is not performance. Its point is to track memory usage. If you need to have your application support the Allocation profiler, you must make sure that AQtime has access to the VCL binary code. The easiest way to add support for the Allocation profiler is to set the **Dynamic RTL** option from the **C++ Linker** category to *False* and then uncheck **Build with runtime packages** in the **Packages** category:



If you want to keep Build with runtime packages (for instance, to control the exe size), you can still use the Allocation profiler. When you include your application in an AQtime project, you will also have to include the <Windows>\System32\RTL100.BPL file. To add a module to an AQtime project, press **Add Module** on the Setup toolbar or select it from the Setup context menu.

13. Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its "debug" version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Note: AQtime is incompatible with some third-party tools that modify the binary code of your application (for example, those that add a custom exception handling mechanism). An example of such a tool is EurekaLog. We recommend that you profile your application before processing it with such tools.

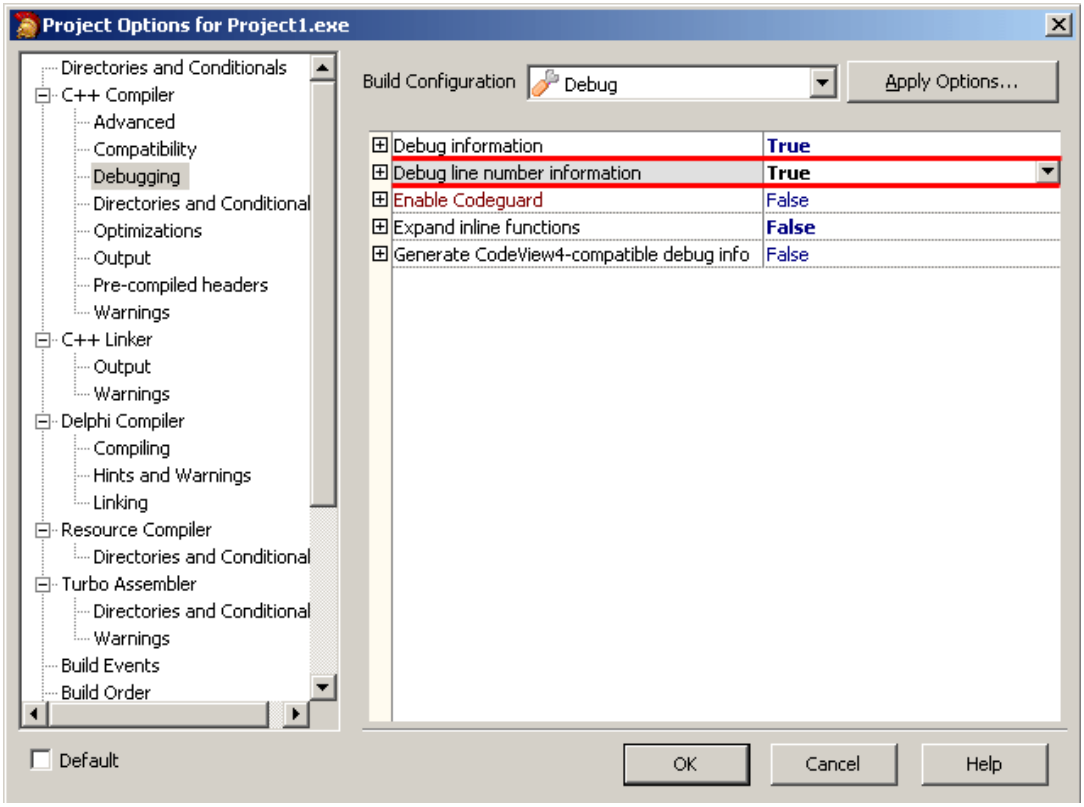
Nevertheless, AQtime is compatible with AQtrace and supports profiling of applications that use AQtrace for error reporting.

Compiler Settings for CodeGear C++Builder 2009

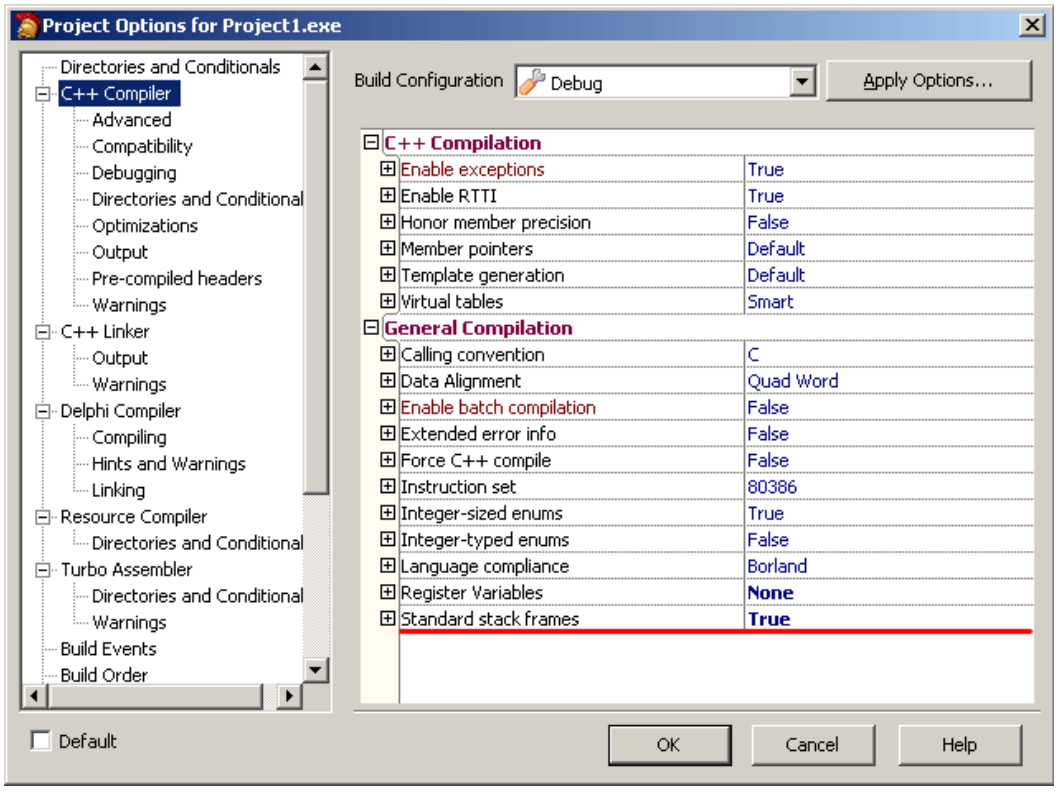
This topic explains how to prepare applications created with CodeGear C++Builder 2009 for AQtime. To learn how to prepare applications created with other C++Builder versions, see *How AQtime Profilers Use Metadata and Debug Information*.

To prepare a CodeGear C++Builder 2009 application for AQtime, you must first ensure that it includes debug info. Follow these steps:

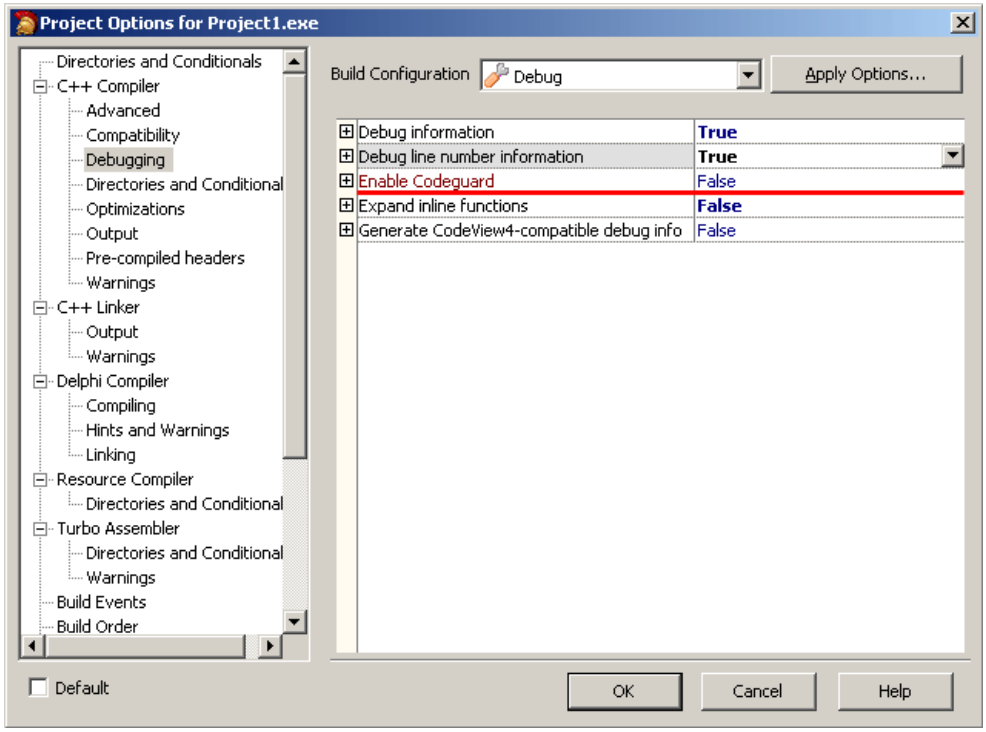
1. Open your project in CodeGear C++Builder 2009.
2. Select **Project | Options** from the main menu. This will open the **Project Options** dialog.
3. To set the C++ compiler options, select the **C++ Compiler | Debugging** category from the tree view on the left of the dialog.
4. To include symbolic debug information, set the **Debug information** option to **True**. In addition, to refer this information to source line numbers, set the **Debug line number information** option to **True**.



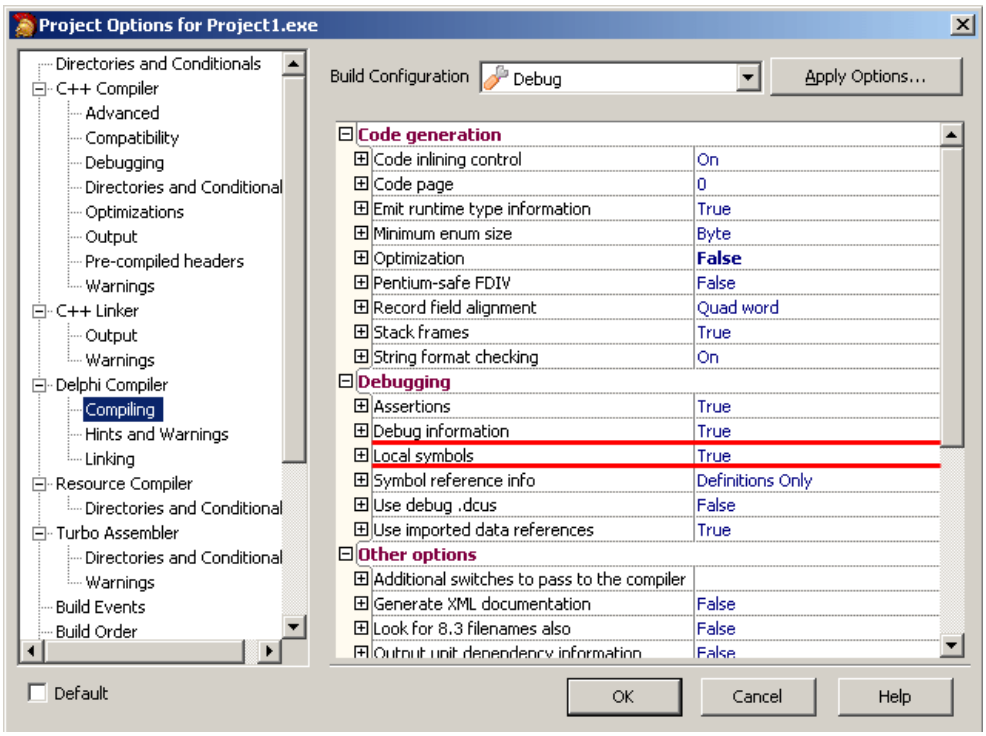
5. (Optional) To generate stack frames when using the C++ compiler, switch to the **C++ Compiler** category and set the **Standard stack frames** option to **True**.



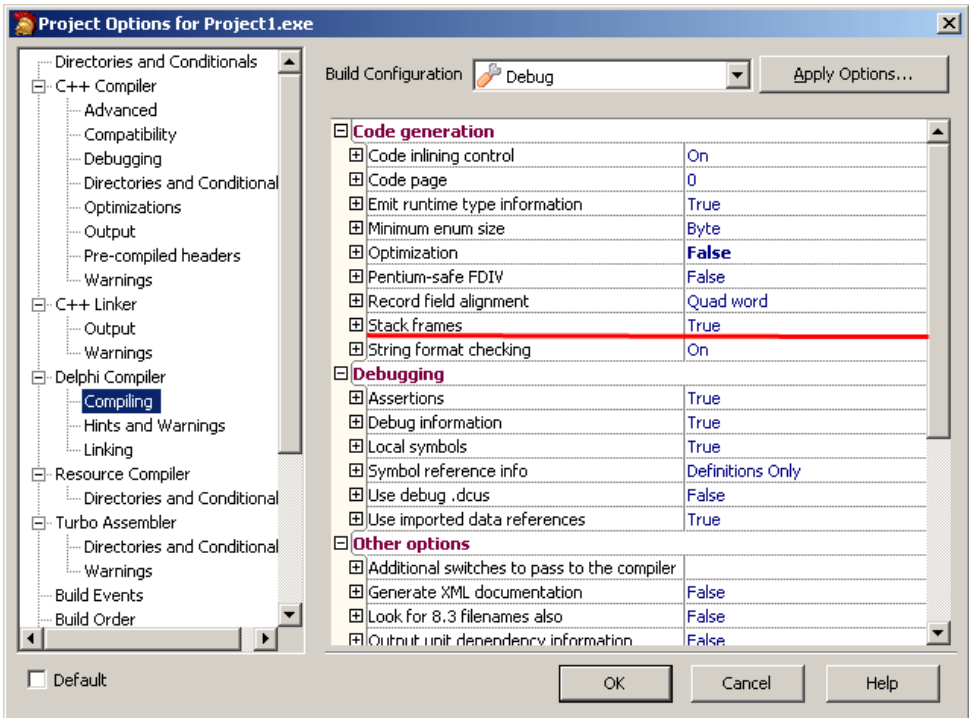
6. If you compile your application with CodeGuard enabled, the latter will report errors during the profiler execution. They relate to AQtime hooking code and are not actually errors. We recommend that you disable CodeGuard before profiling your application with AQtime. To do this, switch to the **C++ Compiler | Debugging** category and set the **Enable CodeGuard** option to **False**.



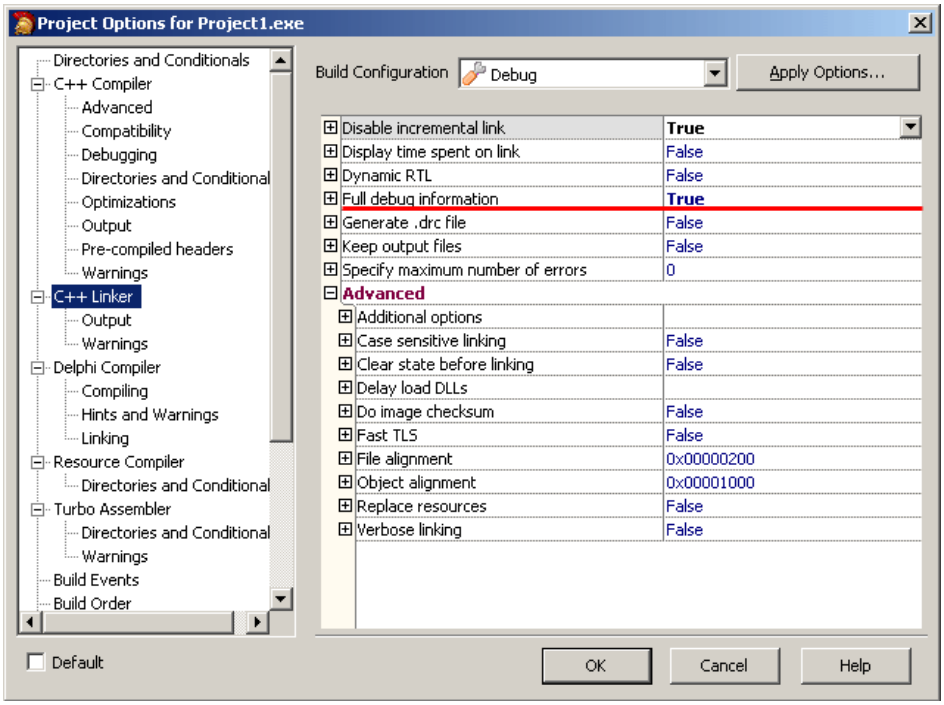
7. To set the Delphi compiler options, select the **Delphi Compiler | Compiling** category from the tree view on the left of the dialog.
8. To include symbolic debug information, set the **Debug information** option to **True**. In addition, to refer this information to source line numbers, set the **Local symbols** option to **True**.



9. (Optional) To generate stack frames when using the Delphi compiler, switch to the **Delphi Compiler | Compiling** category and set the **Stack frames** to **True**.

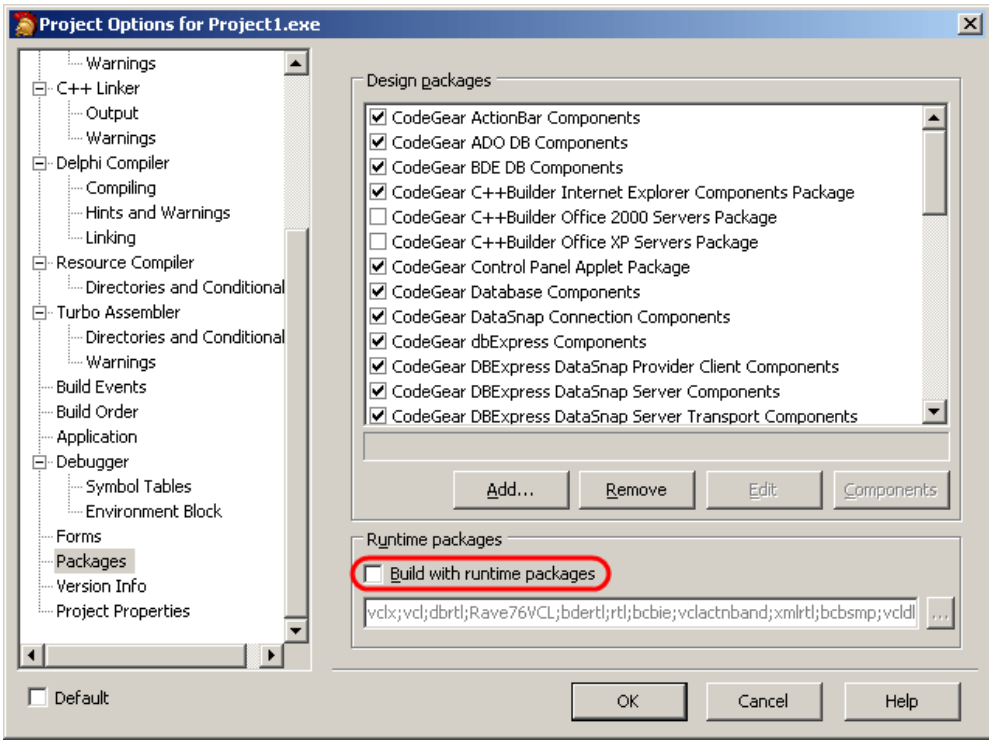


10. To set the linker options, switch to the **C++ Linker** category and set the **Full debug information** option to **True**.



11. Now, if you do not want to use the {*Allocation*} profiler, skip this step.

The point of the Allocation profiler is not performance. Its point is to track memory usage. If you need to have your application support the Allocation profiler, you must make sure that AQtime has access to the VCL binary code. The easiest way to add support for the Allocation profiler is to set the **Dynamic RTL** option from the **C++ Linker** category to **False** and then uncheck **Build with runtime packages** in the **Packages** category:



If you want to keep Build with runtime packages (for instance, to control the exe size), you can still use the Allocation profiler. When you include your application in an AQtime project, you will also have to include the <Windows>\System32\RTL100.BPL file. To add a module to an AQtime project, press **Add Module** on the Setup toolbar or select it from the Setup context menu.

12. Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its "debug" version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Note: AQtime is incompatible with some third-party tools that modify the binary code of your application (for example, those that add a custom exception handling mechanism). An example of such a tool is EurekaLog. We recommend that you profile your application before processing it with such tools.

Nevertheless, AQtime is compatible with AQtrace and supports profiling of applications that use AQtrace for error reporting.

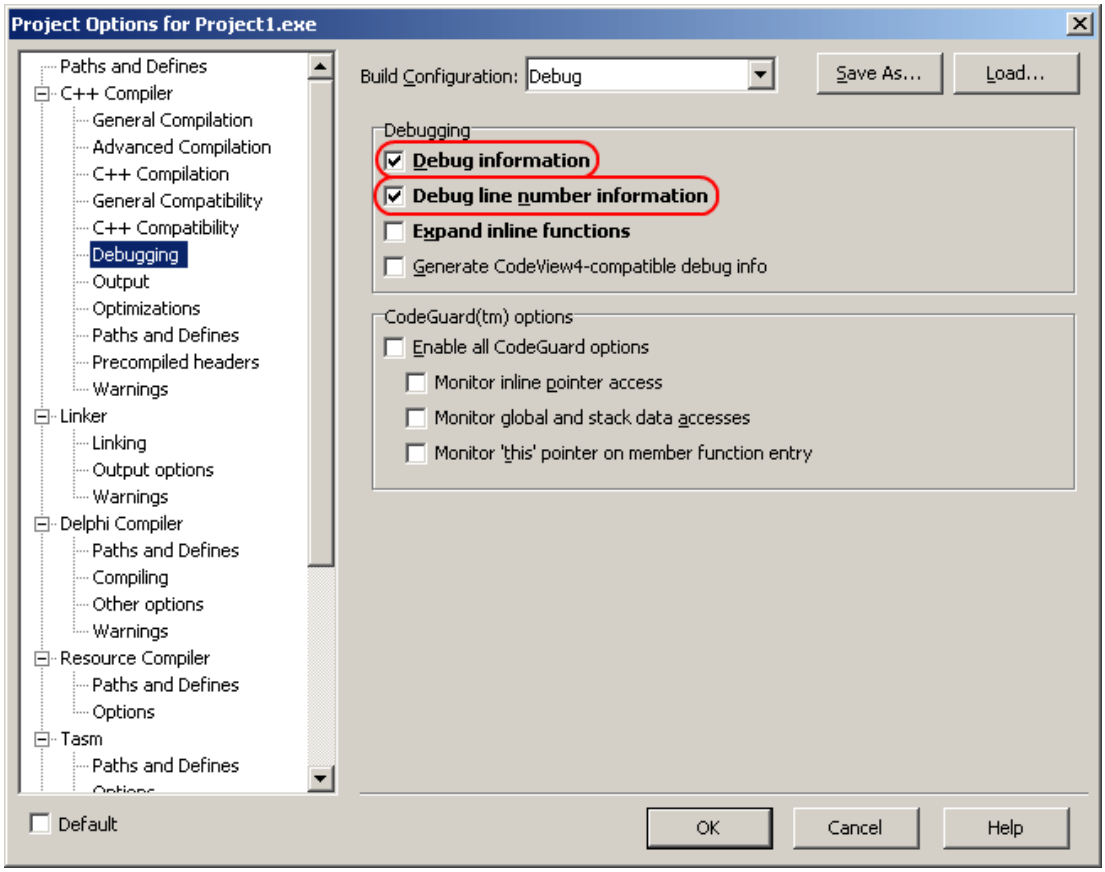
Compiler Settings for CodeGear C++Builder 2007

This topic explains how to prepare applications created with CodeGear C++Builder 2007 for AQtime. To learn how to prepare applications created with other C++Builder versions, see *How AQtime Profilers Use Metadata and Debug Information*.

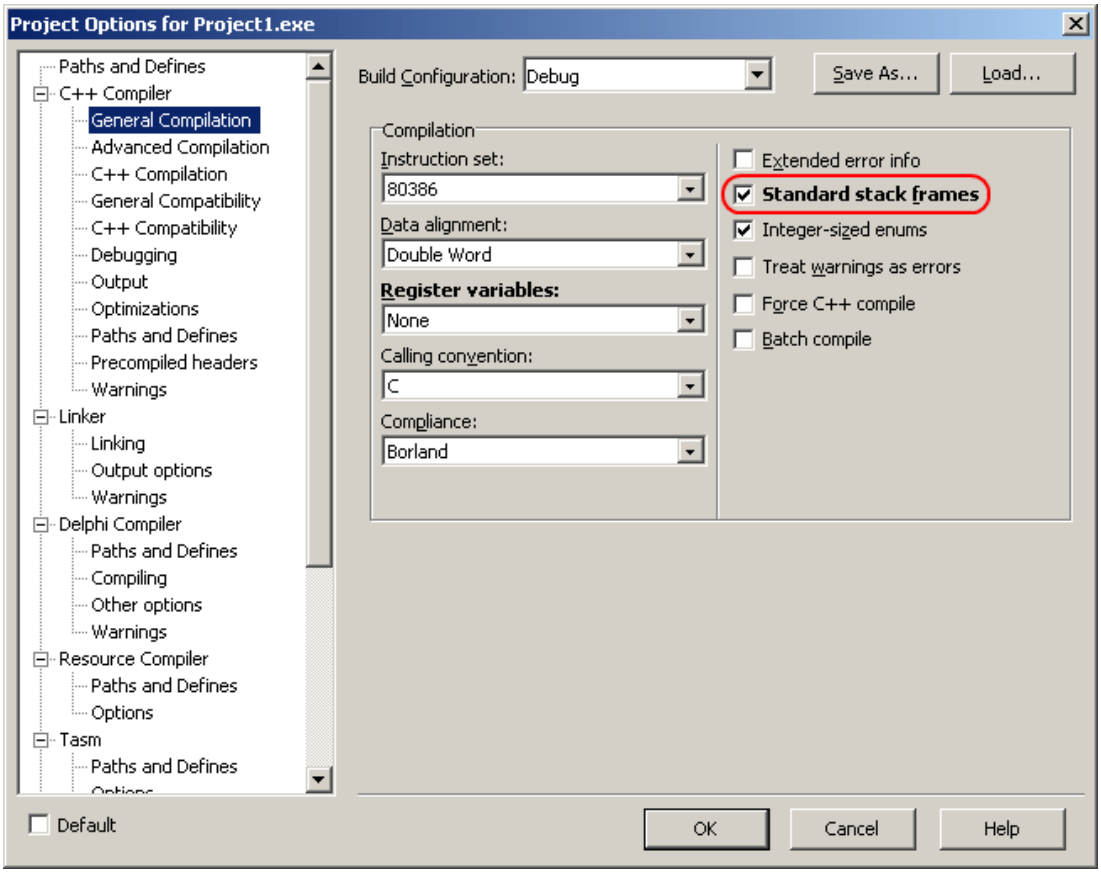
To prepare a CodeGear C++Builder 2007 application for AQtime, you must first ensure that it includes debug info. Follow these steps:

1. Open your project in CodeGear C++Builder 2007.

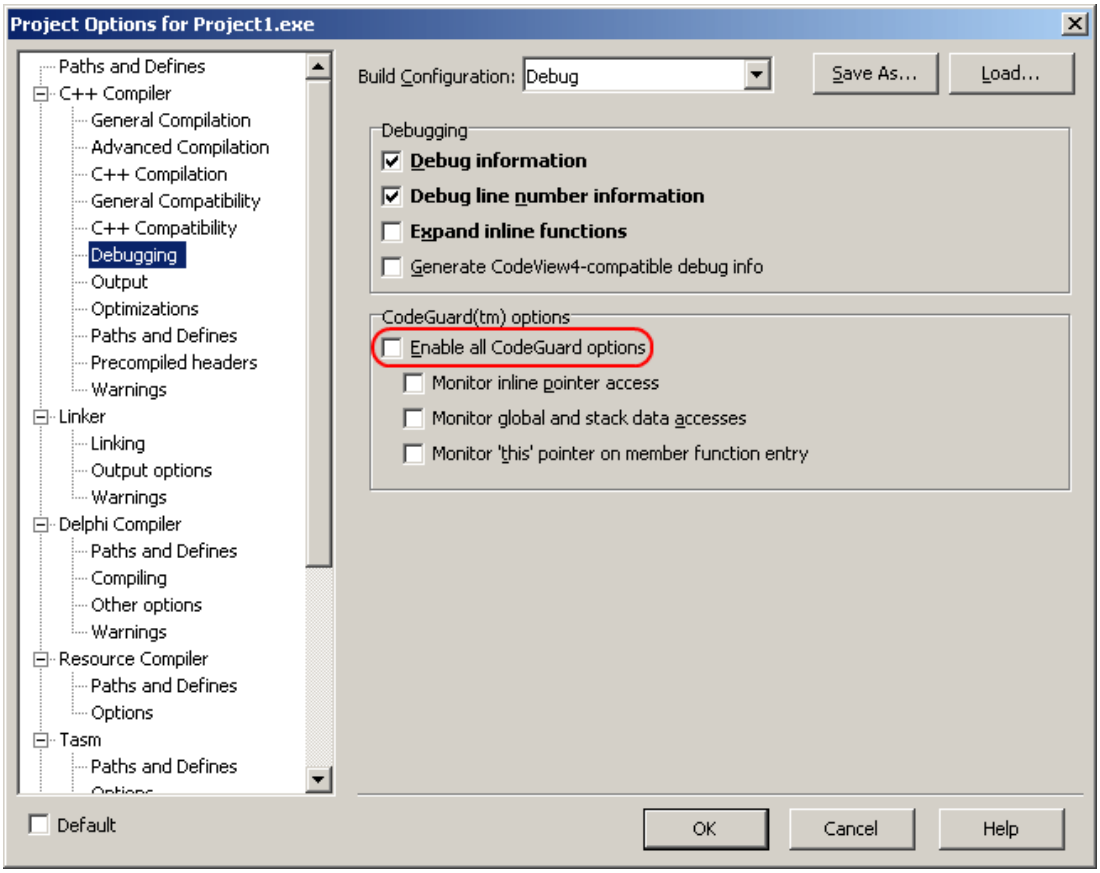
2. Select **Project | Options** from the main menu. This will open the Project Options dialog.
3. To set the C++ compiler options, select the **C++ Compiler | Debugging** category from the tree view on the left of the dialog.
4. To include symbolic debug information, check **Debug information**. In addition, to refer this information to source line numbers, check **Debug line number information**.



5. (Optional) To generate stack frames when using the C++ compiler, switch to the **C++ Compiler | General Compilation** category and check **Standard stack frames**:

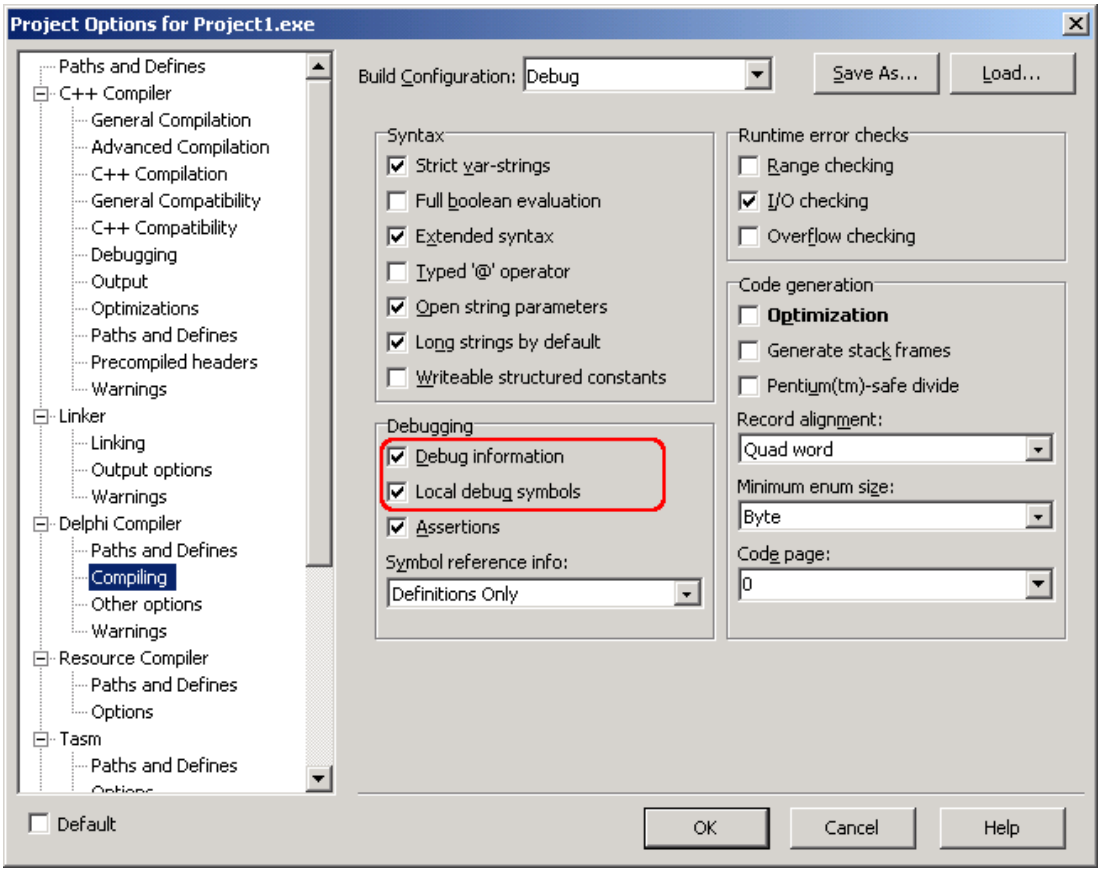


6. If you compile your application with CodeGuard enabled, the latter will report errors during the profiler execution. They relate to AQtime hooking code that are not actually errors. We recommend that you disable CodeGuard before profiling your application with AQtime. To do this, go back to the **C++ Compiler | Debugging** category and uncheck **Enable all CodeGuard options**.

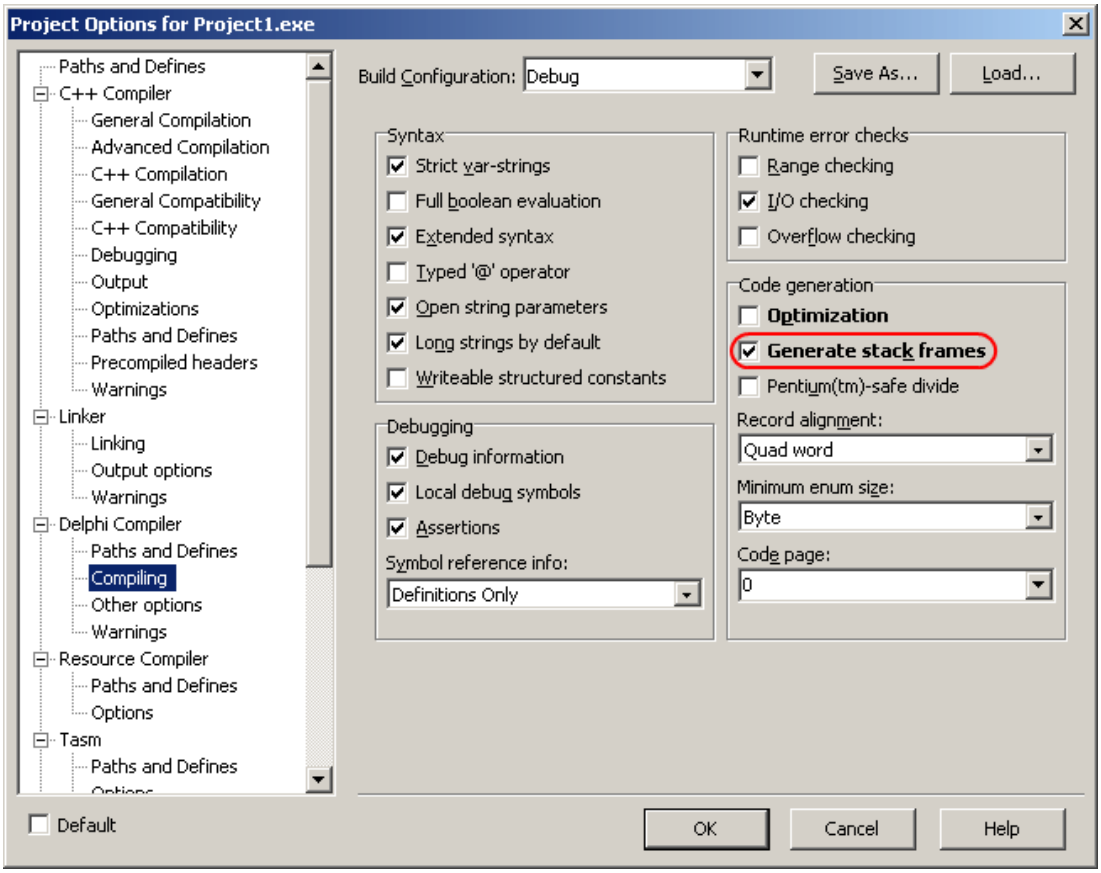


7. To set the Delphi compiler options, select the **Delphi Compiler | Compiling** category from the tree view on the left of the dialog.

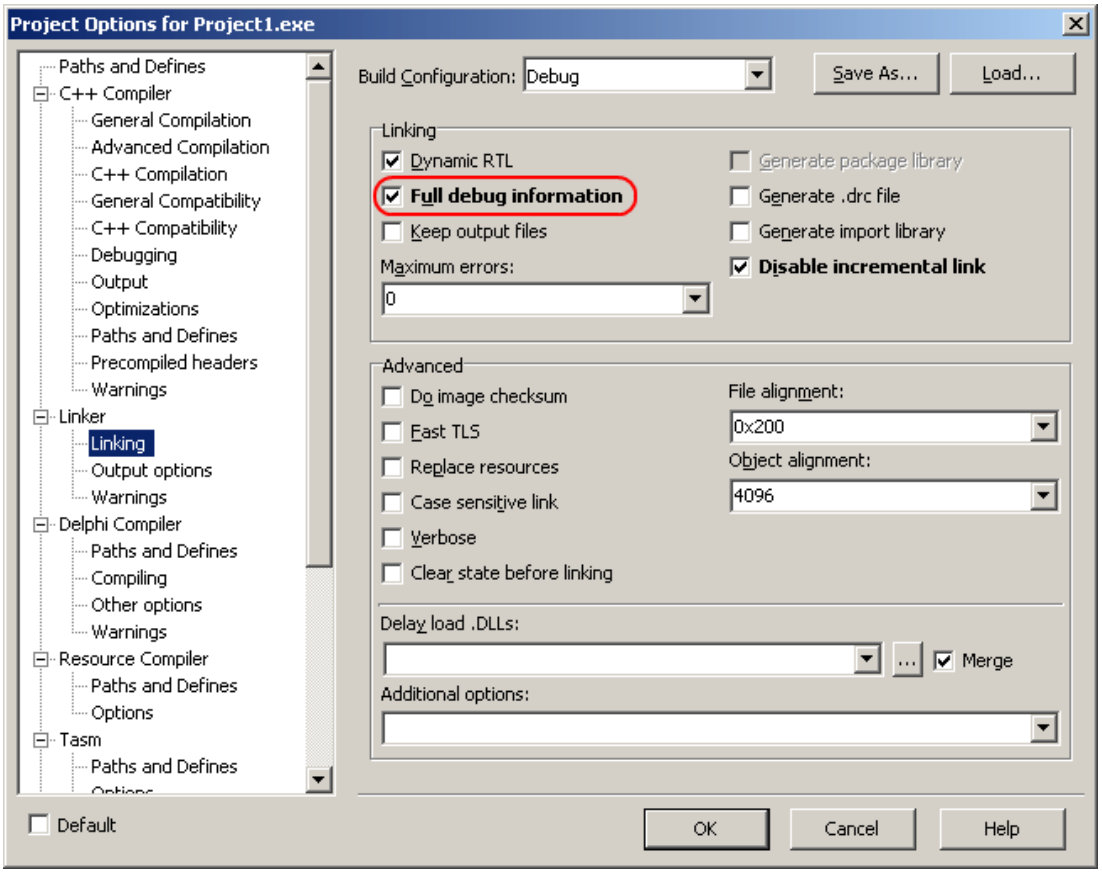
8. To include symbolic debug information, check **Debug information** in the **Delphi Compiler | Compiling** category. In addition, to refer this information to source line numbers, check **Local debug symbols**.



9. (Optional) To generate stack frames when using the Delphi compiler, switch to the **Delphi Compiler | Compiling** category and check **Generate stack frames**.

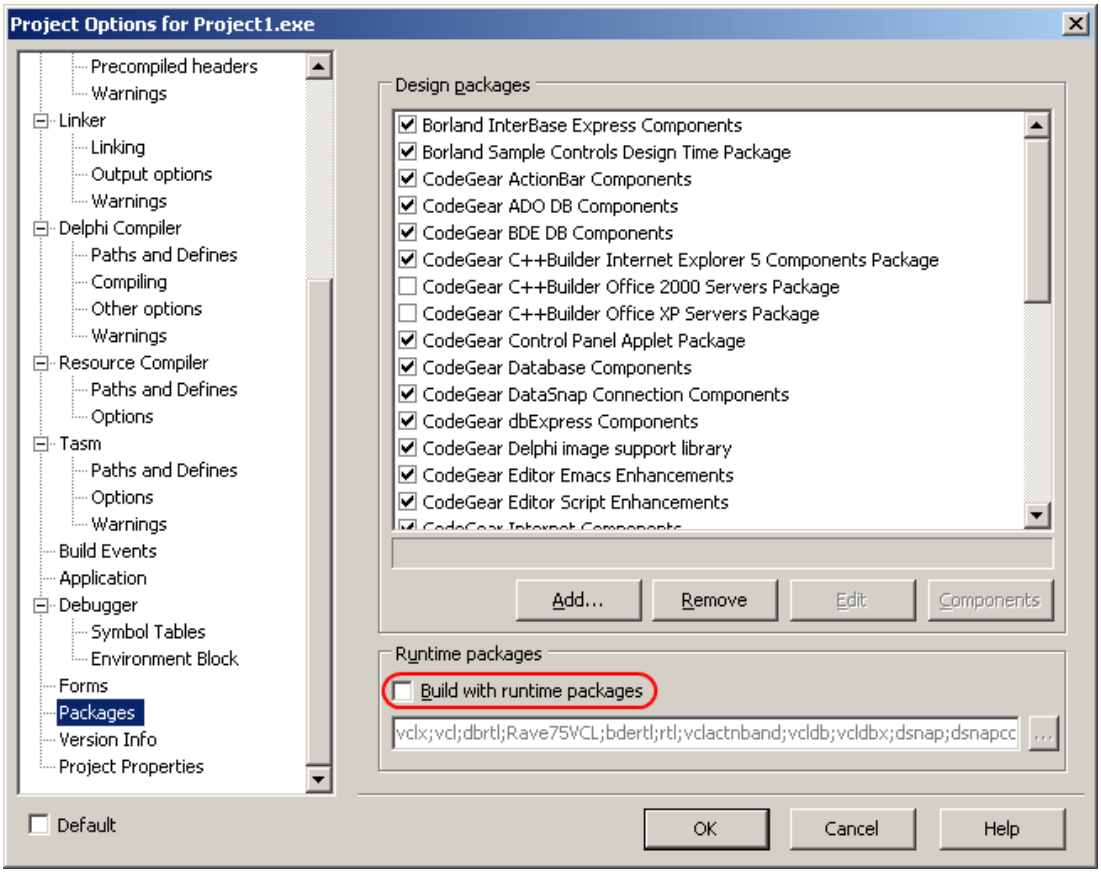


10. To set the linker options, switch to the **Linker | Linking** category and check **Full debug information**.



11. Now, if you do not want to use the Allocation profiler, skip this step.

The point of the Allocation profiler is not performance. Its point is to track memory usage. If you need to have your application support the Allocation profiler, you must make sure that AQtime has access to the VCL binary code. The easiest way to add support for the Allocation profiler is to uncheck **Dynamic RTL** in the **Linker | Linking** category and then uncheck **Build with runtime packages** in the **Packages** category:



Note: If you want to keep the Build with runtime packages option (for instance, to control the exe size), you can still use the Allocation profiler. When you include your application in an AQtime project, you will also have to include the <Windows>\System32\RTL100.BPL file.

To add a module to an AQtime project, press **Add Module** on the Setup toolbar or select it from the Setup context menu.

12. Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its "debug" version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Note: AQtime is incompatible with some third-party tools that modify the binary code of your application (for example, those that add a custom exception handling mechanism). An example of such a tool is EurekaLog. We recommend that you profile your application before processing it with such tools.

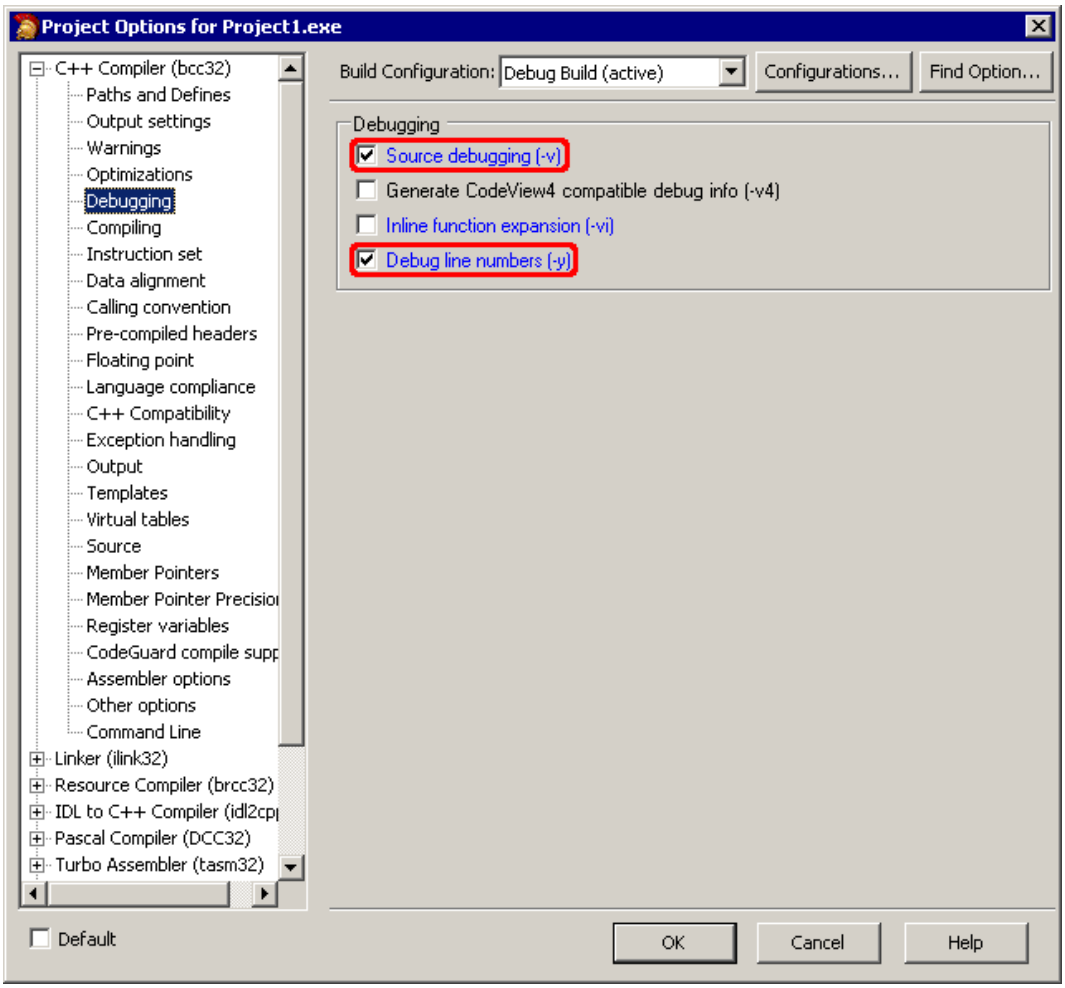
Nevertheless, AQtime is compatible with AQtrace and supports profiling of applications that use AQtrace for error reporting.

Compiler Settings for Borland C++Builder 2006

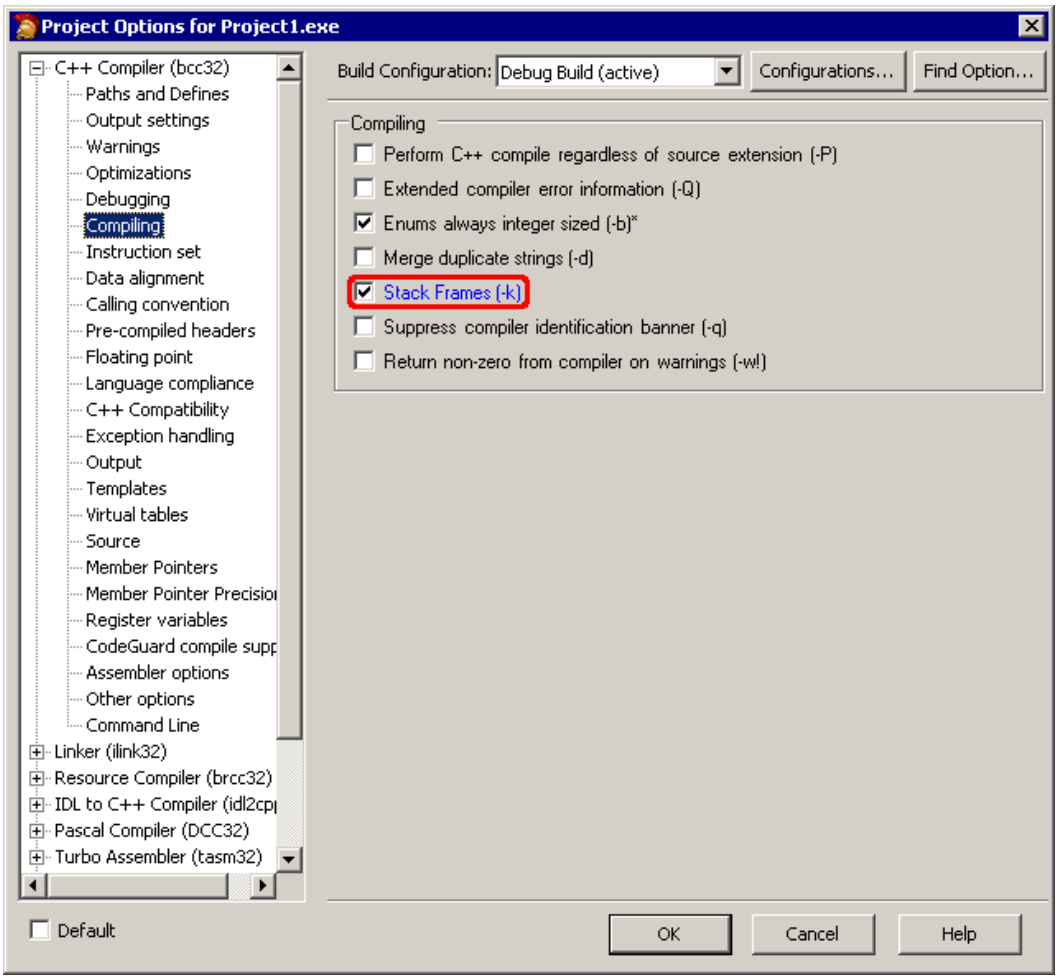
This topic explains how to prepare applications created with Borland C++Builder 2006 for AQtime. To learn how to prepare applications created with other C++Builder versions, see *How AQtime Profilers Use Metadata and Debug Information*.

To prepare a Borland C++Builder 2006 application for AQtime, you must first ensure that it includes debug info. Follow these steps:

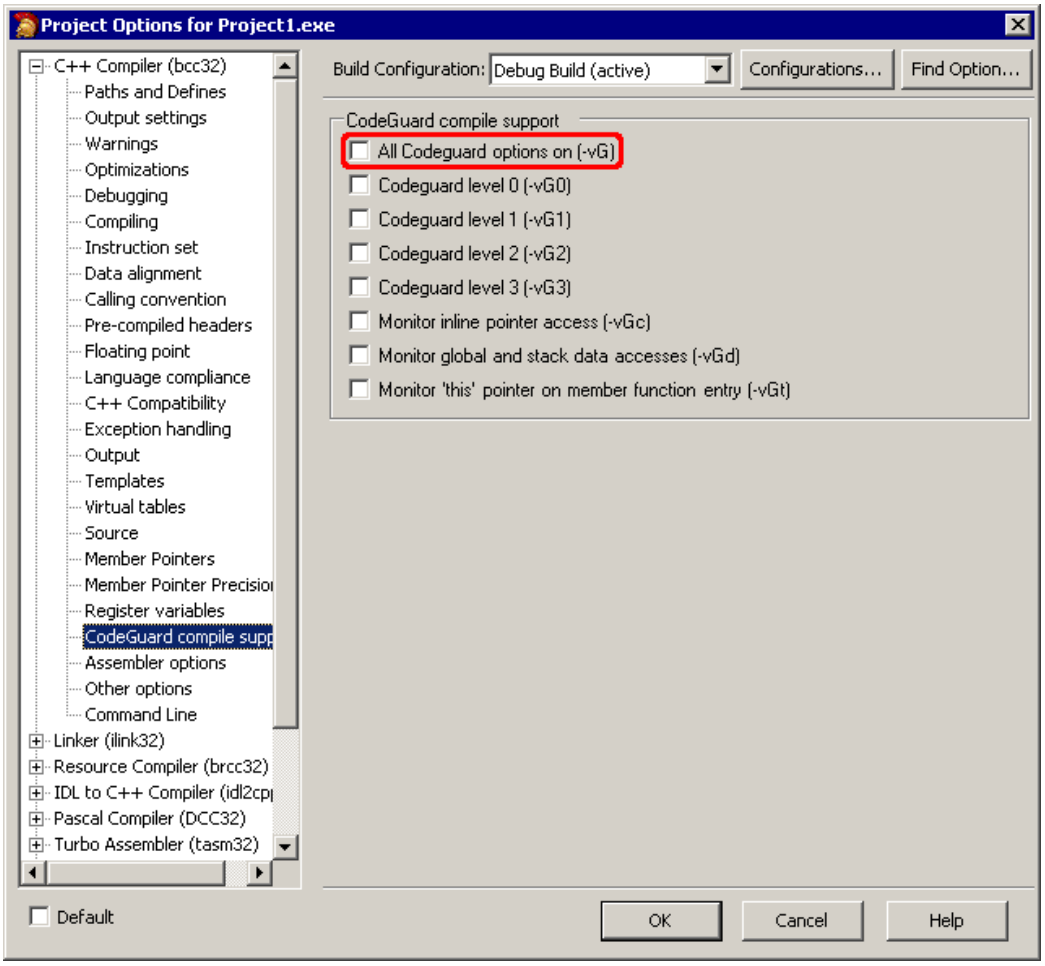
1. Open your project in Borland C++Builder 2006.
2. Select **Project | Options** from the main menu. This will open the **Project Options** dialog.
3. To set the C++ compiler options, select the **C++ Compiler (bcc32) | Debugging** category from the tree view on the left of the dialog.
4. To include symbolic debug information, check **Source debugging**. In addition, to refer this information to source line numbers, check **Debug line numbers**.



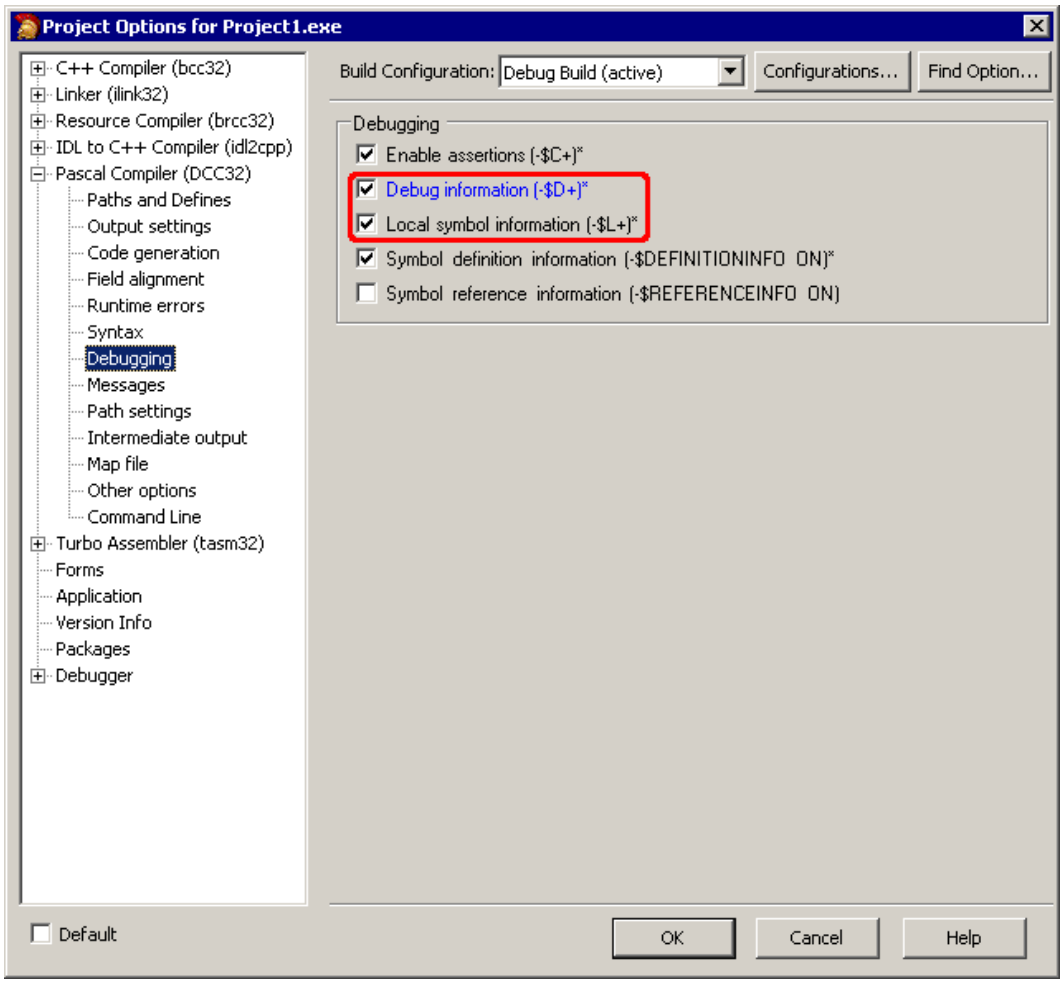
5. (Optional) To generate stack frames when using the C++ compiler, switch to the **C++ Compiler (bcc32) | Compiling** category and check **Stack frames**.



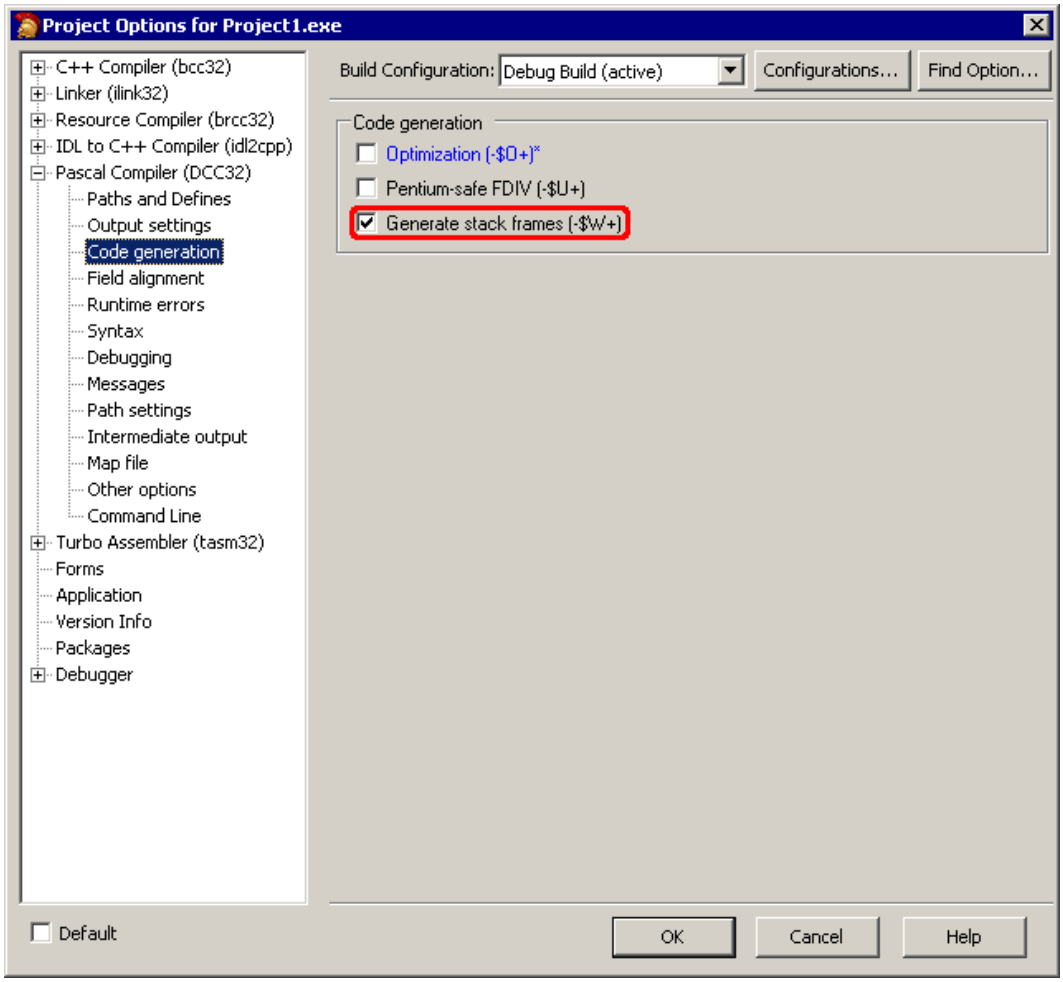
6. If you compile your application with CodeGuard enabled, the latter will report errors during the profiler execution. They relate to AQtime hooking code and are not actually errors. We recommend that you disable CodeGuard before profiling your application with AQtime. To do this, switch to the **C++ Compiler (bcc32) | CodeGuard compile support** category and uncheck **All CodeGuard options on**.



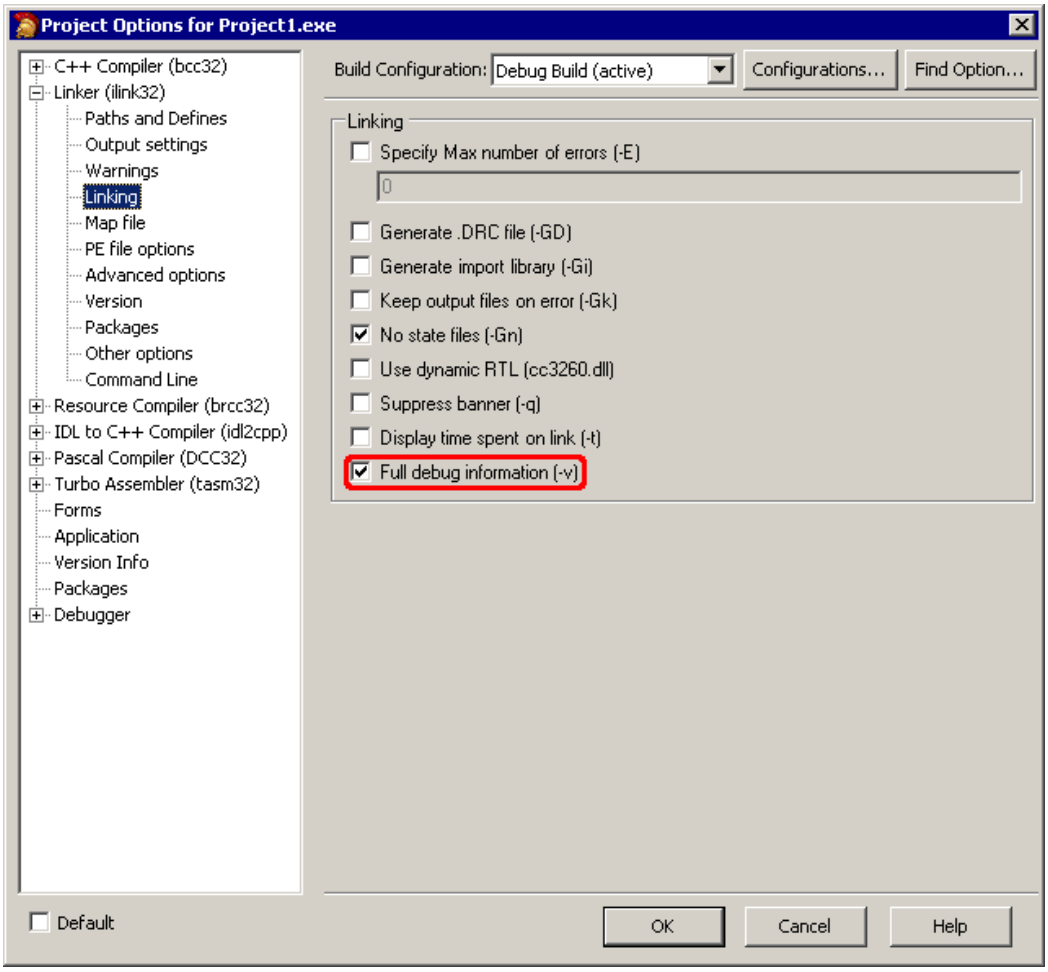
7. To set the Pascal compiler options, select the **Pascal Compiler (DCC32) | Debugging** category from the tree view on the left of the dialog.
8. To include symbolic debug information, check **Debug information**. In addition, to refer this information to source line numbers, check **Local symbol information**.



9. (Optional) To generate stack frames when using the Pascal compiler, switch to the **Pascal Compiler (DCC32) | Code generation** category and check **Generate stack frames**.

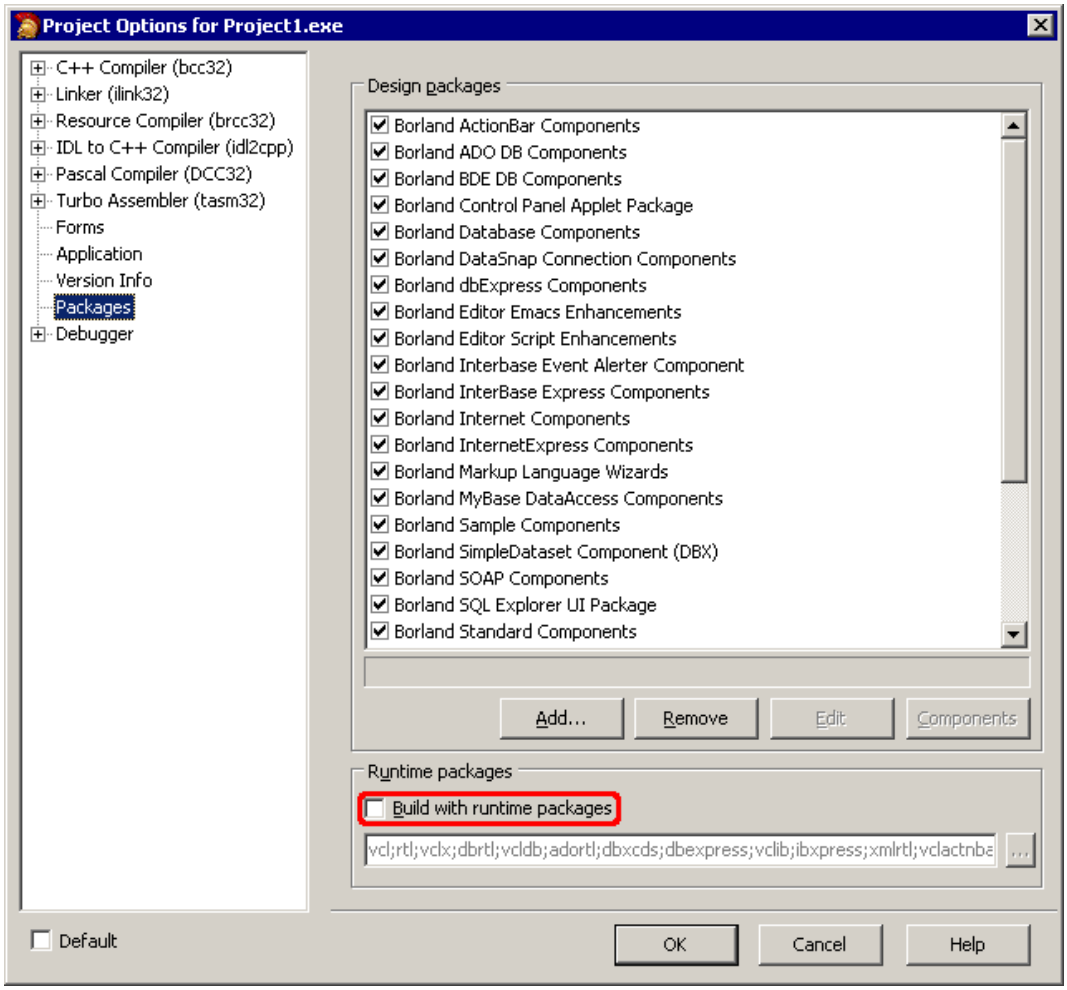


10. To set the linker options, switch to the **Linker (ilink32) | Linking** category and check **Full debug information**.



11. Now, if you do not want to use the Allocation profiler, go to item 12.

The point of the Allocation profiler is not performance. Its point is to track memory usage. If you need to have your application support the Allocation profiler, you must make sure that AQtime has access to the VCL binary code. The easiest way to add support for the Allocation profiler is to uncheck **Use dynamic RTL (cc3260.dll)** in the **Linker (ilink32) | Linking** category and then uncheck **Build with runtime packages** in the **Packages** category:



If you want to keep Build with runtime packages (for instance, to control the exe size), you can still use the Allocation profiler. When you include your application in an AQtime project, you will also have to include the <Windows>\System32\RTL100.BPL file. To add a module to an AQtime project, press **Add Module** on the Setup toolbar or select it from the Setup context menu.

12. Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its "debug" version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Note: AQtime is incompatible with some third-party tools that modify the binary code of your application (for example, those that add a custom exception handling mechanism). An example of such a tool is EurekaLog. We recommend that you profile your application before processing it with such tools.

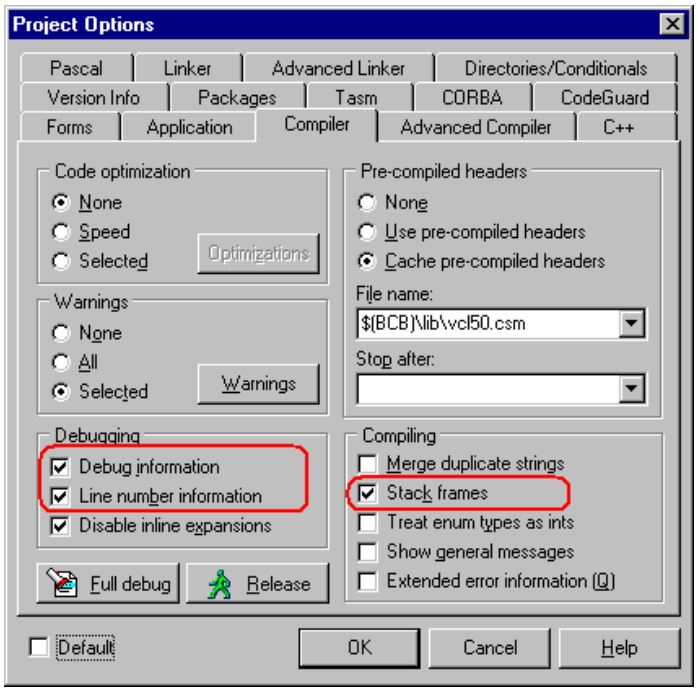
Nevertheless, AQtime is compatible with AQtrace and supports profiling of applications that use AQtrace for error reporting.

Compiler Settings for Borland C++Builder 3-6

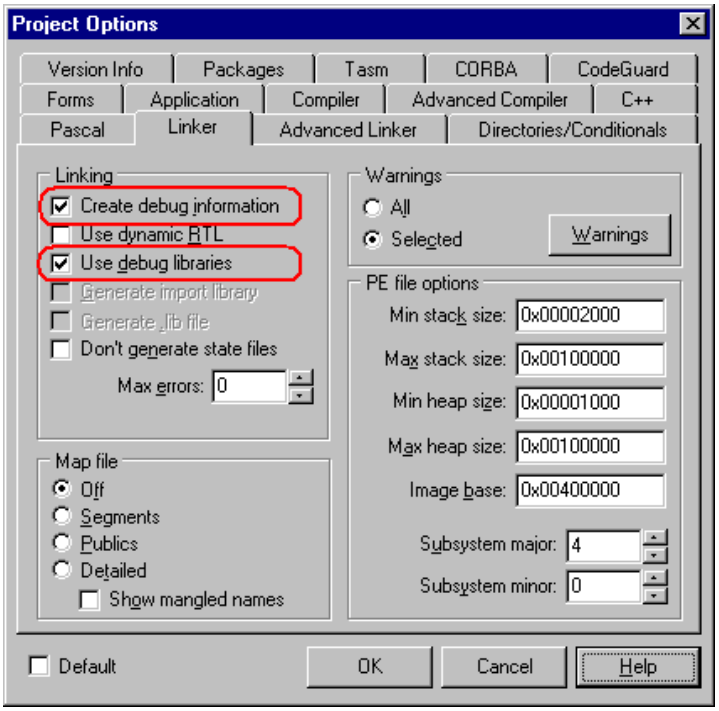
This topic explains how to prepare applications created with Borland C++Builder v. 3-6 for AQtime. To learn how to prepare applications created with other C++Builder versions, see *How AQtime Profilers Use Metadata and Debug Information*.

To prepare a Borland C++Builder application for AQtime, you must first ensure that it includes debug info. Follow these steps:

1. Open your project in Borland C++Builder.
2. To set the compiler options, choose **Project | Options** from C++Builder's main menu and select the **Compiler** tabbed page.
3. To include symbolic debug information, in the **Debugging** section of the **Compiler** page, check **Debug information**. In addition, to refer this information to source line numbers, check **Line Information**.

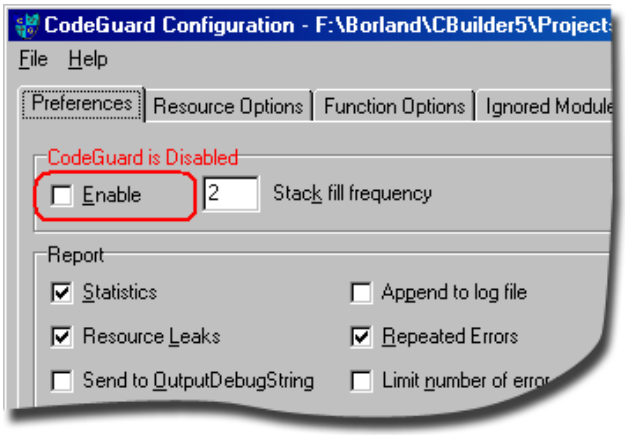


4. To set the linker options, select the **Linker** tabbed page. In the **Linking** options group, check **Create Debug Information**.
5. If you want to profile VCL classes, for example, `TDataset`, check the **Use debug libraries** box. Otherwise, AQtime will be able to profile only the classes that are defined in your application.



6. If you compile your application with CodeGuard enabled, the latter will report errors during the profiler execution. They relate to AQtime hooking code and are not actually errors. We recommended that you disable CodeGuard before profiling your application with AQtime. To do this:

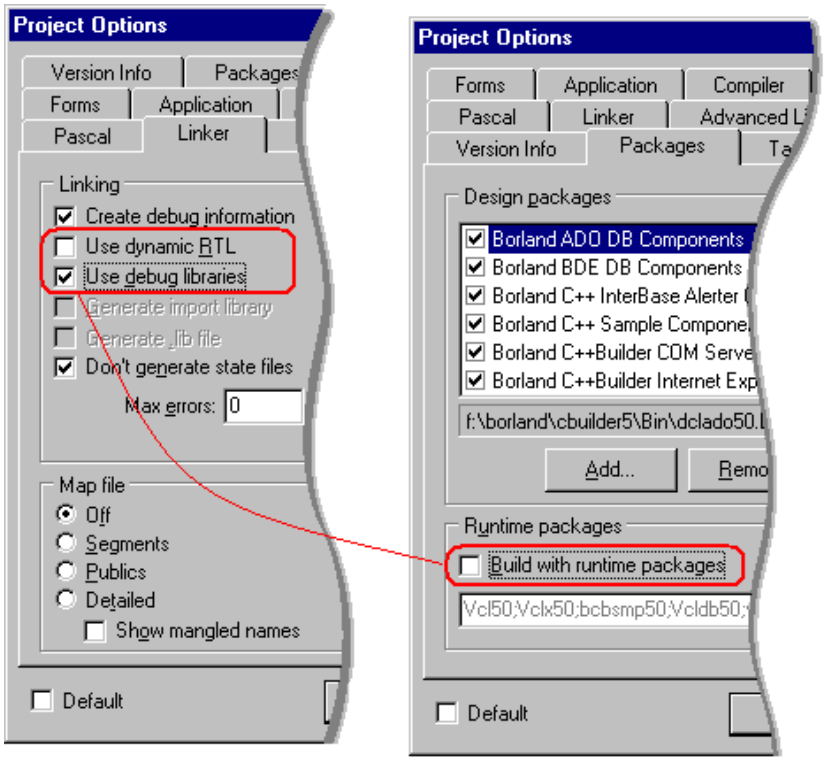
   - Select **Tools | CodeGuard Configuration** from C++Builder's main menu. This will open the **CodeGuard Configuration** dialog.
   - Uncheck the **Enable** box on the **Preferences** tabbed page of this dialog:



   - Click **OK** to save the changes.

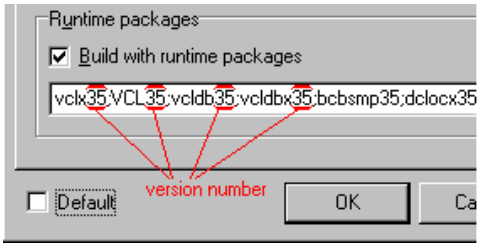
7. Now, if you do not want to use the Allocation profiler, go to point 8.

   The point of the Allocation profiler is not performance. Its point is to track memory usage. If you need to have your application support the Allocation profiler, you must make sure that AQtime has access to the VCL binary code. The easiest way to add support for the Allocation profiler is to check **Use debug libraries** on the **Linker** page, uncheck **Use dynamic RTL** on that page, and then uncheck **Build with runtime packages** on the **Packages** page:



Tip You can actually leave the **Use dynamic RTL** checked, but in this case the memory would be allocated via the *Borland C++ Multi-thread Runtime Library* (cc3260mt.dll) and since this module is not included in the Setup panel, all found leaks would not be displayed, if the *Show leaks for Setup modules* option is enabled. This option is enabled by default and the results of the Allocation profiler would not be shown until the option is disabled, which could be rather confusing.

If you want to keep the Build with runtime packages enabled (for instance, to control the exe size), you can still use the Allocation profiler. When you include your application in an AQtime project, also include the <Windows>\System32\VCL*nn*.BPL or <Windows>\System32\RTL*nn*.BPL file, where *nn* is the compiler version number. You can find this number in the edit field under the "Build with runtime packages" check box:



For instance:

- Add VCL35.BPL if you use C++Builder 3.
- Add VCL50.BPL if you use C++Builder 5.
- Add RTL60.BPL if you use C++Builder 6.

To add a module to an AQtime project, press **Add Module** button on the Setup

toolbar or select it from the Setup context menu.

8. Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its "debug" version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

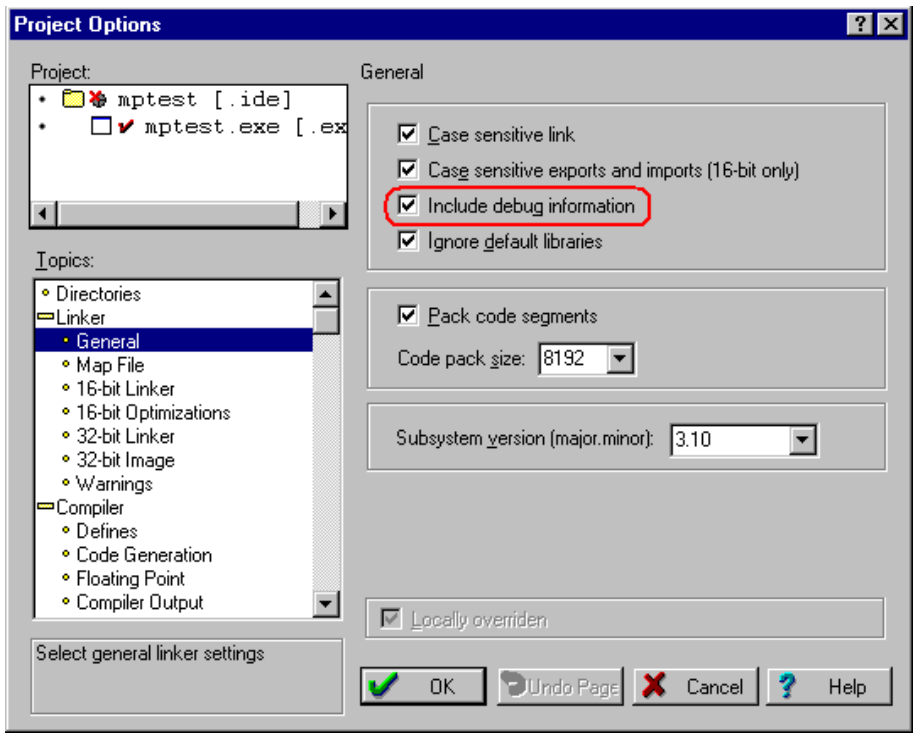
Note: AQtime is incompatible with some third-party tools that modify the binary code of your application (for example, those that add a custom exception handling mechanism). An example of such a tool is EurekaLog. We recommend that you profile your application before processing it with such tools.

Nevertheless, AQtime is compatible with AQtrace and supports profiling of applications that use AQtrace for error reporting.

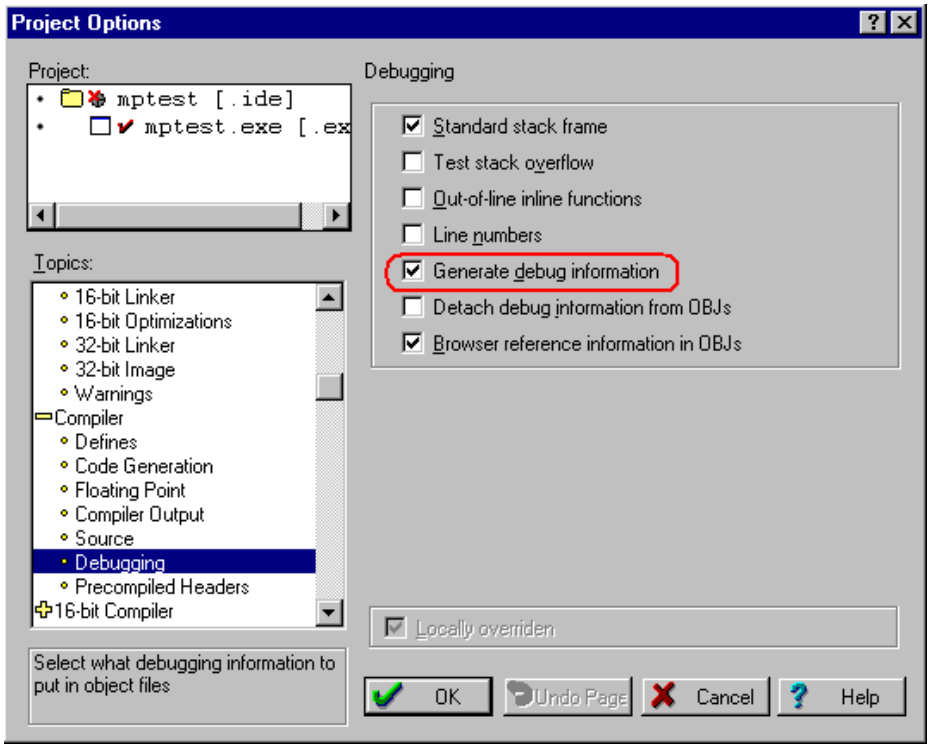
Compiler Settings for Borland C++

To prepare a Borland C++ application for AQtime, you simply need to ensure that it includes debug info. Follow these steps:

- To set the compiler options, choose **Project | Options** from Borland C++'s main menu and select the **Compiler** topic.
- To include the symbolic debug information, from the Compiler topic, select the **Debugging** subtopic. Once there, check **Generate Debug Information**:



- To set the linker options, from Project | Options now select the **Linker** topic. In the **Linking** options group, check **Create Debug Information**:



Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an ActiveX control or a COM server however, you should register its "debug" version in the system (See *Profiling COM Applications*).

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Compiler Settings for Intel C++

The Intel C++ 7.0 compiler is tightly integrated with Microsoft Visual Studio 6.0 and 7.0. To prepare an Intel C++ application for AQtime, you should perform the same actions, which you perform to prepare a Visual C++ application. For complete information, review the following topics:

- *Compiler Settings for Visual C++ 6.0*
- *Compiler Settings for Visual C++ .NET*

Compiler Settings for GNU CC

The current AQtime version must get debug information in the **stab** (symbol table) format, so you must use the GDB extension for stab.

To generate debug information in stab format, use either the `-g` or `-ggdb` compiler option:

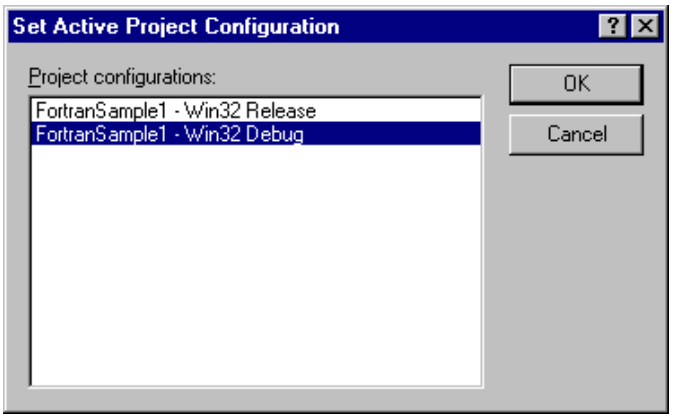
- `g` means "debug information in the format native to the operating system", and stab is OS-native for Cygwin.
- `ggdb` means "debug information in the format native to the compiler", and again this is stab for Cygwin.

Remember that future versions of GCC may use a different default debug information format. For more information on GCC arguments controlling the creation of debug information, see the *Options for Debugging Your Program or GNU CC* topic of the GCC Compiler Guide.

Compiler Settings for Compaq Visual Fortran

To prepare a Visual Fortran application for AQtime, you must ensure that it includes the debug info and select the format under which it will be generated. Follow these steps:

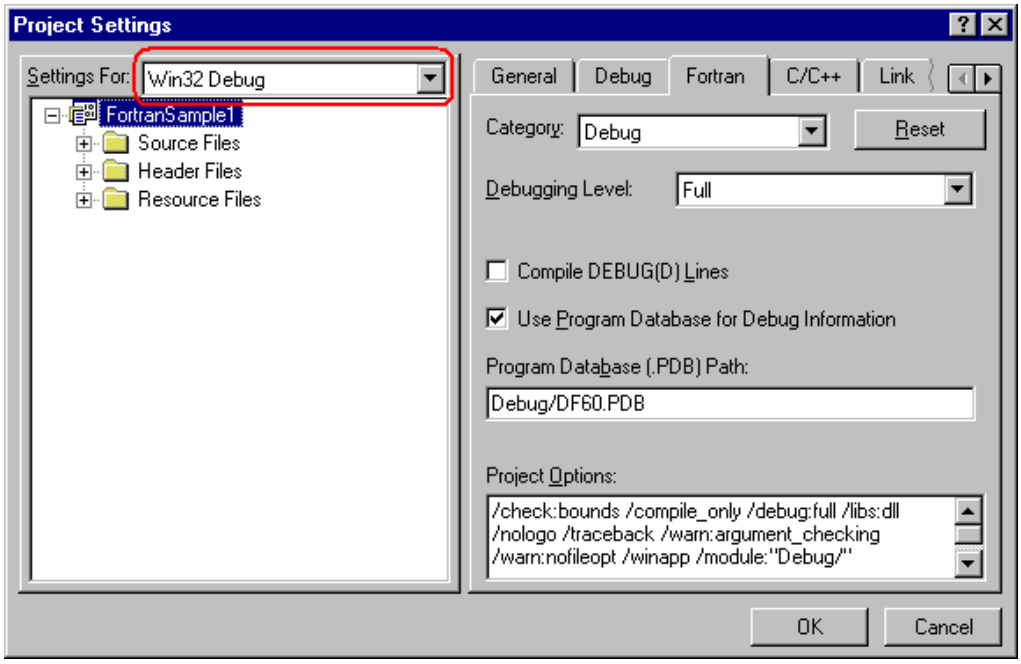
1. Open your application in Visual Fortran's IDE.
2. Select **Build | Set Active Configuration** from the main menu. This will open the following dialog:



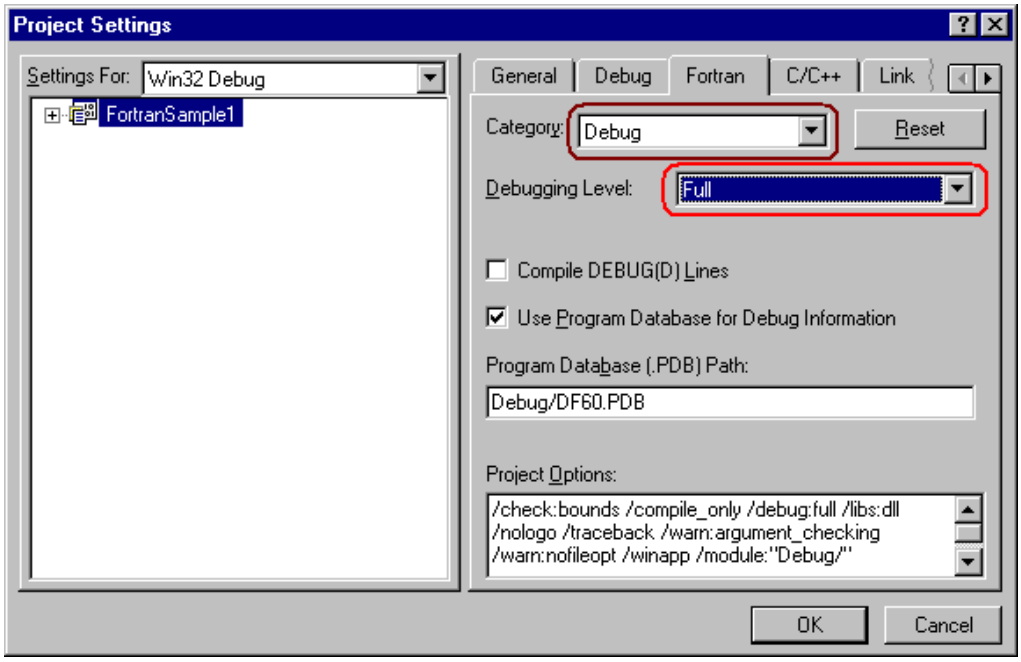
In the dialog, set the *<Your project name> - Win32 Debug* configuration as active and press **OK** to save changes.

Note that the only reason to change the active configuration is to leave the *Release* configuration unaffected. If you want to profile the *Release* configuration, you may skip this step. In this case, do not forget to restore the compiler settings before compiling the release version of your application.

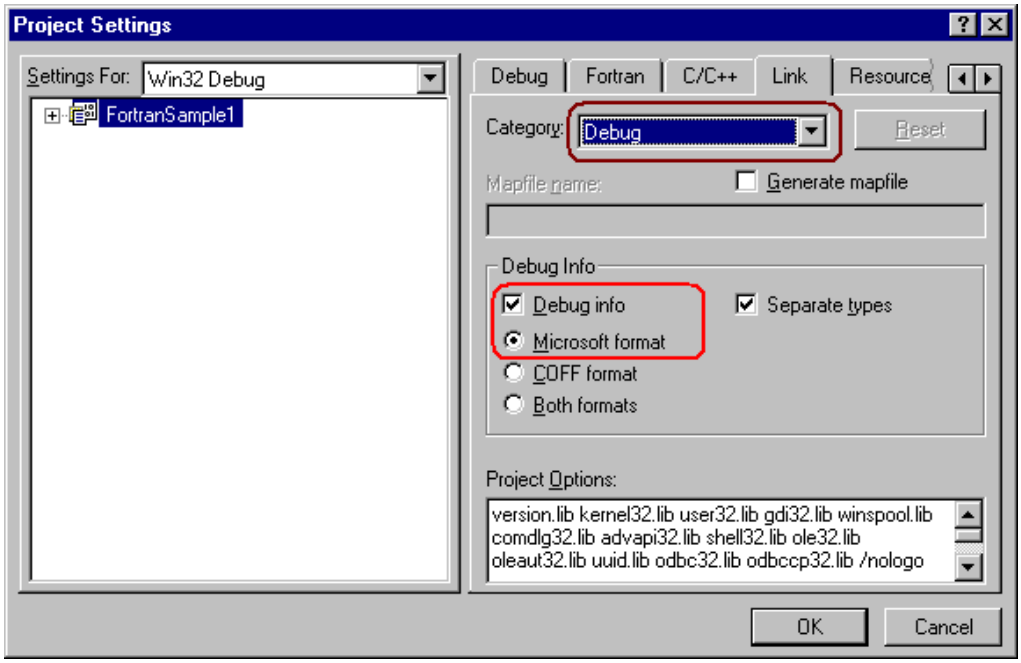
3. Select **Project | Settings** from the main menu. The **Project Settings** dialog will appear. Select *Win32 Debug* in the **Settings for** box.



4. Switch to the **Fortran** tabbed page and select *Debug* from the **Category** box. Then, set *Full* in the **Debugging Level** field:

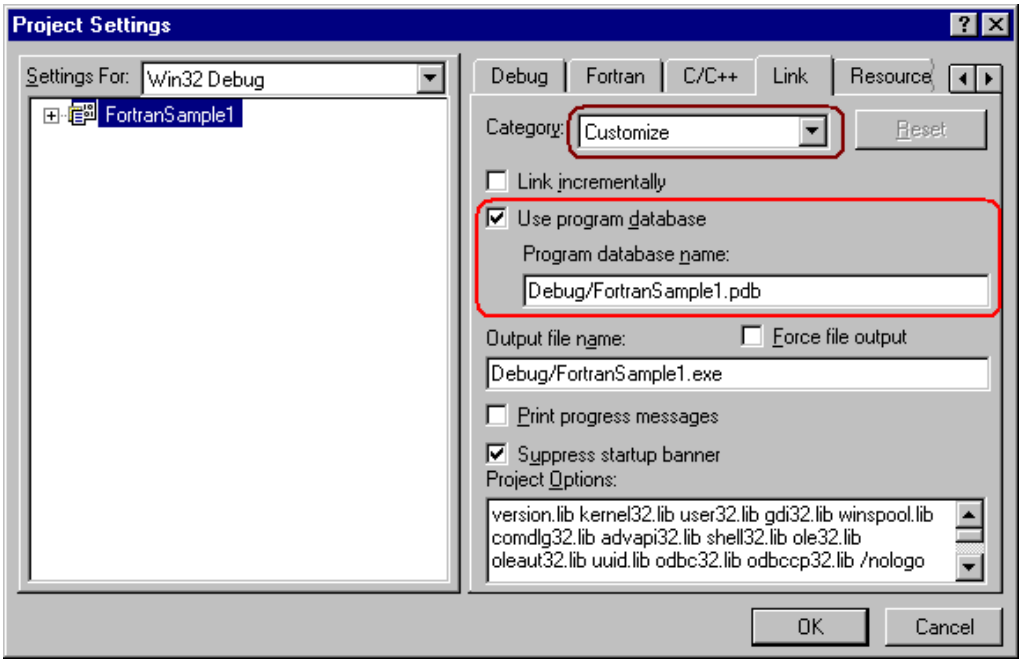


5. Switch to the **Link** page and set **Category** to *Debug*. Now check the **Debug info** box and select **Microsoft format of the debug information**:



6. The last step is to set how the linker will save the debug information. AQtime supports debug information generated as an external PDB file (PDB format). To set linker options:
   - Switch to the **Link** page of the **Project Settings** dialog.
   - Set the **Category** to *Customize*.
   - Check **Use program database**.

- Enter the PDB file name you want into the **Program database name** edit field.



7. Press **OK** to close the Project Settings dialog. Recompile your application.

By default, the .pdb debug info file resides in the same folder where the profiled executable or DLL is located. When you open your module in AQtime, it searches for the debug info file in this default lcoation. If you copy the debug info file to another location, then specify this location in the **Symbols Path** list of the Symbols Options dialog (this list contains the search paths). See *Specifying Path to Debug Info Files* for more information.

When your application is ready for release, remember to recompile it without debug information to reduce the application size.

Setting up the Project

Opening a Project

Your AQtime **project** is simply your current "work site" in AQtime. The project specifies the application and modules to profile, profiling parameters, profiling mode, etc. The project file also holds recent profiling results, which you can permanently save. So the project is also your set of available past results.

If you use AQtime as a standalone application, then to create a new project, select **File | New Project** from AQtime's main menu. AQtime will create a new project.

To create create a new project in AQtime integrated into Visual Studio .NET 2002, Visual Studio .NET 2003 or Visual Studio 2005, follow these steps:

- Select **File | New | Project** from Visual Studio's menu or press the **New Project** button on the Start page. This will call the **New Project** dialog.
- In the dialog:
  - Select **AQtime Projects** from the list on the left of the dialog and then click AQtime Project on the right.

    - Specify the project name, location and solution.
- Press OK.

Visual Studio will create a new AQtime project and display its contents in the Solution Explorer.

In AQtime integrated into Borland Developer Studio, AQtime projects (.aqt files) are part of the AQtime project groups (.bdsproj files), which are simply containers for several AQtime projects. So, before creating a new AQtime project you must first create and open an AQtime project group to which this project will belong. To create a new AQtime project group in Borland Developer Studio:

- Right-click somewhere in the **Project Manager** panel and select **Add New Project** from the context menu, or select **File | New | Other** or **Project | Add New Project** from Borland Developer Studio's menu. This will call the **New Items** dialog.
- In the dialog, select **Profiling** from the list on the left of the dialog, click **AQtime Project Group** on the right and click **OK**.

Borland Developer Studio will create a new AQtime project group and display its contents in the Project Manager.

To create a new AQtime project in the opened AQtime project group:

- Select **File | New | Other** from Borland Developer Studio's menu. This will call the **New Items** dialog.
- In the dialog, select **Profiling** from the list on the left of the dialog, click **AQtime Project Module** on the right and click **OK**.

Borland Developer Studio will add a new AQtime project to the opened AQtime project group and display its contents in the Project Manager.

Once you have created a new project, you can add the modules (EXE, DLL, OCX, etc.) you wish to profile to the project. To add a new module, select **Add Module** from the context menu of the **Setup** panel or from the **Setup** (**Setup.AQtime**) toolbar and select the desired file using the subsequent Open File dialog. This will work in both the standalone and integrated versions of AQtime. In Visual Studio and Borland Developer Studio you have one more way to add modules to your project: right-click the project in the Solution Explorer or in the Project Manager respectively and select **Add Module** from the context menu. To add a .NET assembly registered in the Global Assembly Cache (GAC) to the AQtime project, select **Add Assembly** from the context menu of the Setup panel or from the Setup (Setup.AQtime) toolbar; or right-click the project in Visual Studio's Solution Explorer or in Borland Developer Studio's Project Manager and select **Add Assembly** from the context menu. This will call the **Add Assembly** dialog, where you can select the desired assemblies. To add the web page whose script you want to profile, select **Add URL** from the context menu of the Setup panel or from the Setup (Setup.AQtime) toolbar and enter the page address in the ensuing Add URL dialog.

If you add an AQtime project to an existing solution in Visual Studio, you can use the **Add Project Output** dialog to choose which output modules of projects that exist in the solution you want to add to the newly created AQtime project. To call this dialog, right-click your AQtime project in the Solution Explorer and then select **Add | Add Output** from the context menu.

An AQtime project can only contain 32-bit (x86) or 64-bit (x64) modules, but not both. The project cannot contain modules with different "bitness". This behavior is caused by a Windows limitation that 32-bit modules can be loaded into a 32-bit process only and 64-bit modules can be loaded into 64-process only (you cannot load a 32-bit module to a 64-bit process). So, if you add a 64-bit module to your AQtime project, you can continue adding 64-bit modules only. If you add a 32-bit module, you can only add 32-bit modules. To change the "bitness" of the project, clear the modules list and then add the desired modules.

You can add as many modules, from as many folders, as you wish. The module that was added to the AQtime project first, will be the *main* module. This means that AQtime will launch this module when you start profiling. Other modules are not started by AQtime; they will be loaded by the main module. You can change the main module at any time by right-clicking the desired module and selecting **Set as Active Module** from the context menu. If the main module is a DLL or an OCX file, you have to specify a Host Application for your project so that AQtime can start profiling. The host application can be set in the Run Parameters dialog.

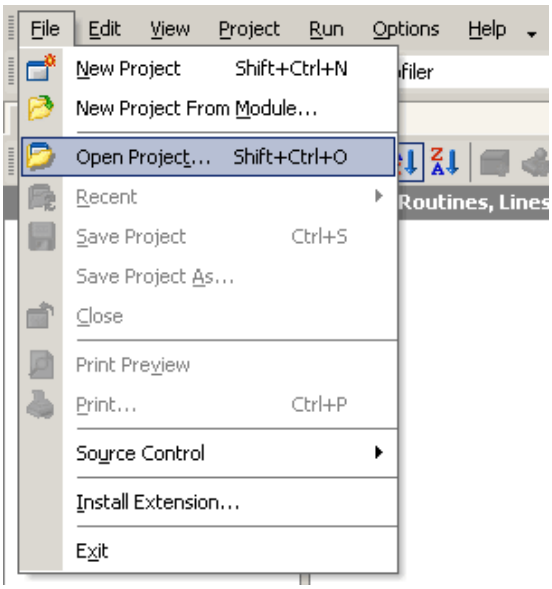
An alternative way to create a new project is to select **File | New Project From Module** from AQtime's main menu. This will call the Open File dialog where you can select an executable (EXE, DLL, OCX) which AQtime will use to create the new project. Once you press **Open** in this dialog, AQtime will automatically create a new project that will contain the module you have selected. It will be the main module of the project. After you have created a project, you can add modules to it as it is described above.

To save your new project, select **File | Save** from the main menu. This will open the standard Save File dialog where you can specify the project file name. If you create a new project and do not save it, the results and profiling configuration will be lost upon closing the project. However, once you have saved a project to a file, the results and changes you make to the profiling configuration will be saved automatically (AQtime saves them upon closing the project or after you select the File | Save menu item).

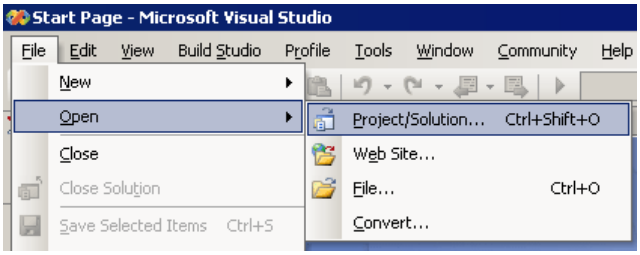
By default, AQtime project files have the .aqt extension. To save an existing project under another name or extension, select **File | Save As** from the main menu.

If you use AQtime standalone, then after you have saved your project to a file, you can add this file to a source control system like Visual SourceSafe or CVS. AQtime is tightly integrated with source control systems, so you can add your project to a source control directly from AQtime. For complete information on how to do this, see *Working with Source Control Systems*. If you use AQtime integrated in Visual Studio, you can put your project in Visual SourceSafe using Visual Studio's means.

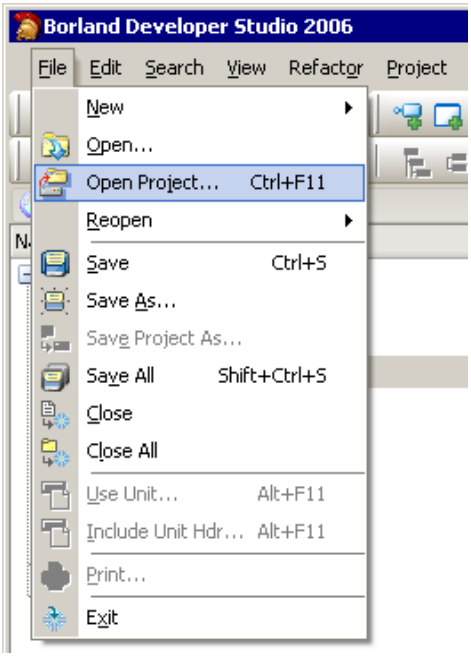
To open an existing project in AQtime, select **File | Open Project**:



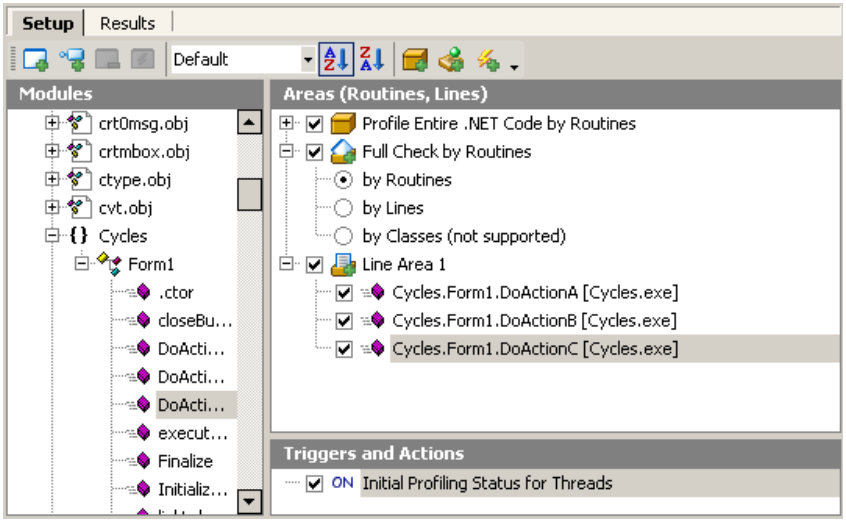
To open an existing AQtime project in Visual Studio, just select **File | Open | Project\Solution** (that is, you open AQtime projects in the same manner as you open any other Visual Studio project):



To open an existing AQtime project in Borland Developer Studio, just select **File | Open Project** (that is, you open AQtime projects in the same manner as you open any other Borland Developer Studio project):



Upon selecting the Open Project menu item, AQtime will display the standard Open File dialog where you can select the desired project file name. Once your project is open in AQtime, you can see a list of all object modules and their routines in the left-hand pane of the Setup panel:



Controlling What to Profile

Profilers yield a mass of information. The trick in using them is to ask only for the kind of information you need at the moment, get it, then start over again, using what you've just learned to refine the question you are asking. In other words, you **dig down** progressively.

Therefore, even more important than **what** information a profiler can provide is how easily it will **focus** on what you want to know. A good profiling tool is one that you can ask very restricted questions easily, get answers, then re-tune your question. Otherwise, the important information gets lost in the mass of results which, at the present moment, are of no importance or, worse, a distraction. (In addition, a few profilers seriously add to execution time, so you don't want to wait while they gather information you won't need).

A lot of the advantages of AQtime reside in the ease of use, variety and flexibility of the means it provides you with for **controlling** what gets profiled in any given run. All of them work on the **exclusion** principle: if a given means says something will not be profiled, it will not be. If it does not say that, or it says the code "will" be profiled, then the code will only actually be profiled if all the other means permit. From the very general to the very local, these means group into the following categories:

- Filter non-modifiable code. Many development environements include a number of their libraries in your application. For example, Borland Delphi IDE typically embeds System, Classes, Controls and other units. Generally, the source code of these libraries cannot be modified, so their performance cannot be improved. Therefore, you should focus only on those elements that you can change. To filter out modules provided by standard libraries, you can use AQtime's *Exclude standard source files* option. The option can also be enabled via the Exclude Standard Source Files button located on the Setup panel.
- Define **code areas to profile**. Profiling areas are a central concept in AQtime. Any number of files, classes or routines can be included in an area, and any number of areas can be checked (or unchecked) for profiling in a given run. Furthermore, each element in a checked area can also be checked or unchecked. Areas are a primary tool for progressive refining of what you want to profile. As noted, code correctly checked-in this way only gets profiled if all of the other means permit it. But what is not checked does not get profiled on this run, period. Because sometimes you may want to put an entire class or namespace in an area, except one or two elements, in addition to the normal including areas there are excluding areas. Since nothing gets profiled if it is not in an including area, the point of excluding areas is only this, to provide an easy way of removing certain sub-elements from a larger element added to an including area. See *Defining Areas to Profile* and *Checking Elements to Profile*.
- Define when to profile your code. **Triggers** are another central concept in AQtime. They apply only to the Performance, Function Trace and Coverage profilers and they let you control profiling on a thread by thread basis. There are on-triggers and off-triggers. An on-trigger is a routine that turns profiling on when it begins and turns it off (unless another trigger is running) when it ends. Code that is correctly checked in the area system, and not excluded as a "system file", will be profiled only when it is called from a trigger routine in the same thread, directly or indirectly. Off-triggers are the opposite. While they are running, whatever profiling would be going on in their thread is turned off. If there are no trigger routines, then profiling is always on by default (the application is the trigger). See *Using Triggers* and *Setting up Triggers*.
- Turn profiling on and off during the run. The **Enable/Disable Profiling** toolbar button (the **Profile | Enable/Disable Profiling** menu item in Visual Studio or the **Enable/Disable Profiling** button on Borland Developer Studio's **Run.AQtime** toolbar) can turn profiling off at any time while the application is running. When it is "on" (the default), profiling is enabled (of course, it is enabled only if no off-trigger or action is active). When this button is not pressed, the profiling is off. This is a really quick, no-fuss, no-mess way to restrict profiling to a given trouble spot -- once

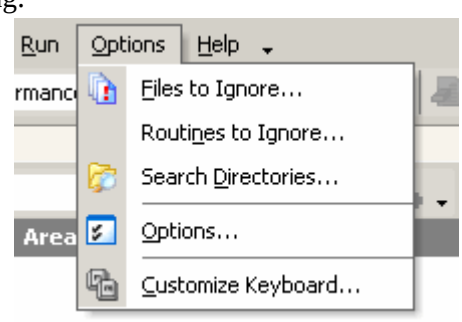
you know where it occurs. Its drawback is that you can never repeat the run exactly; for run-to-run comparisons, actions or triggers are the tools to use.

- Perform specific operations during the run: **actions**. An action is a routine, before or after execution of which AQtime can perform a specific operation like switching the profiling on or off or getting the profiler results. Actions are similar to pressing the **Enable/Disable Profiling** or the **Get Results** toolbar items from the application code. But unlike manual pressing, actions allow you to press these items exactly when you need it. For more information on actions and differences between triggers and actions, see *Using Actions*.

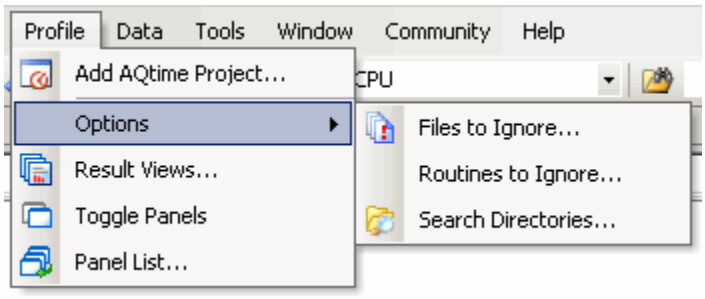
Excluding Code From Profiling

All the means listed in *Controlling What to Profile* are actually means of excluding code from profiling, or of restricting profiling to certain times during the run of the application. The object of this topic is the means of excluding files or functions that you will "never" want to profile, either in any application, or in the current project. In other words, we are talking about exclusions that are global AQtime settings, or entire-project settings. More-controlled exclusions are better defined through the areas facility.

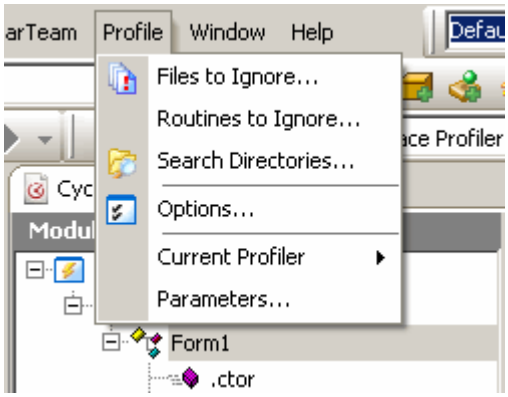
The first two items of the **Options** menu in AQtime running as a standalone application (the **Profile | Options** menu in Visual Studio and the **Profile** menu in Borland Developer Studio) let you specify files and routines to be excluded from profiling:



Options menu in AQtime standalone



Options menu in AQtime integrated into Visual Studio



Options menu in AQtime integrated into Borland Developer Studio

Item one is **Files to Ignore** and item two is **Routines to Ignore**. The point of Files to Ignore is that there are files you might wish to profile, but you also want to skip some of the large functions which clog up the profiling results, and that you do not normally want or need to profile.

Both of these settings are global to AQtime, they work equally for all projects. There are times where you might wish that they were not so restrictive. Rather than try to undo and redo them, you can use the Bypass ignore settings option, which is normally off but can easily be turned on -- and then back off to restore normal behavior. If you choose Ignore on a serious project, make sure that you enable other means of restricting what gets profiled. See *Controlling What to Profile*.

You can also define files and routines to exclude from the current project only. This is done through the Files to Ignore item of the **Project** menu (see *Files to Ignore for Project Dialog*).

One more way to exclude files from profiling is to use the Exclude routines with no source info option. The debug information may not hold information about source files for some routines. If this option is enabled, these types of routines become “invisible” to AQtime services. They are excluded from profiling, they are not shown in the Setup panel, and so on.

There are some cases when the excluding settings are ignored:

- A routine that is added to an action or trigger is treated as a routine that is added to a profiling area. These routines are not profiled if profiling is off during the routines’ execution. For more information, see *Using Acitons* and *Setting Up Triggers*.
- All your settings that exclude managed code from profiling are ignored, if the *Profile Entire .NET Code* box is checked in the Setup panel. Therefore, if you add a managed routine to the Routines to Ignore dialog and *Profile Entire .NET Code* is checked, the routine will still be available for profiling. More information about the *Profile Entire .NET Code* box you will find at one of the next steps.

Profiling Levels

AQtime profilers can analyze application code at three levels of detail: routine, line or class. The level of profiling is specified by the **Level** property of the profiling area (See *Defining Areas to Profile*):

- Routine and line level areas are supported by the Performance and Coverage profilers only:
- If a routine was added to a **routine level** area, AQtime gathers information for the entire routine. For example, the Performance profiler will trace how many times the routine was called, how long it worked, etc.
- If the routine was added to a **line level** area, AQtime gathers information for each line of source code within it. For instance, for each source line, the Performance profiler will measure execution

time, Hit Count value, etc. If you add a class or module to the line level area, all methods of this class or module will be profiled at line level.

Line profiling requires information about lines in source code. That is why to profile a .NET application at the line level, you must compile it with debug information. Metadata is not enough (See *How AQtime Profilers Use Metadata and Debug Information* for details).

- The **class level** profiling areas are supported by the Allocation profiler only. The profiler tracks the creation and deletion of objects whose class names were added to these areas. If you add a file or module to a class level area, then AQtime will track all objects whose classes are declared in this file or module.

Defining Areas to Profile

AQtime offers several means of **restricting** what parts of an application get profiled. **Profiling areas** (with the associated **checking**) are perhaps the most important of these means. But they work in association with the other means. In the end, what gets profiled is what currently falls under no restriction of one kind or another. See *Controlling What to Profile* for full details.

But first note that some profilers can ignore areas and checking, for example, the Exception Tracer or a third-party profiler. They always profile the entire project. The rest of this topic only applies to the Performance, Coverage, Light Coverage, Allocation and Function Trace profilers.

Once a project is open (that is, an application has been selected for profiling, see *Opening a Project*), the Setup panel is where you set and change most parameters that will apply across profilers. These are areas, triggers and actions.

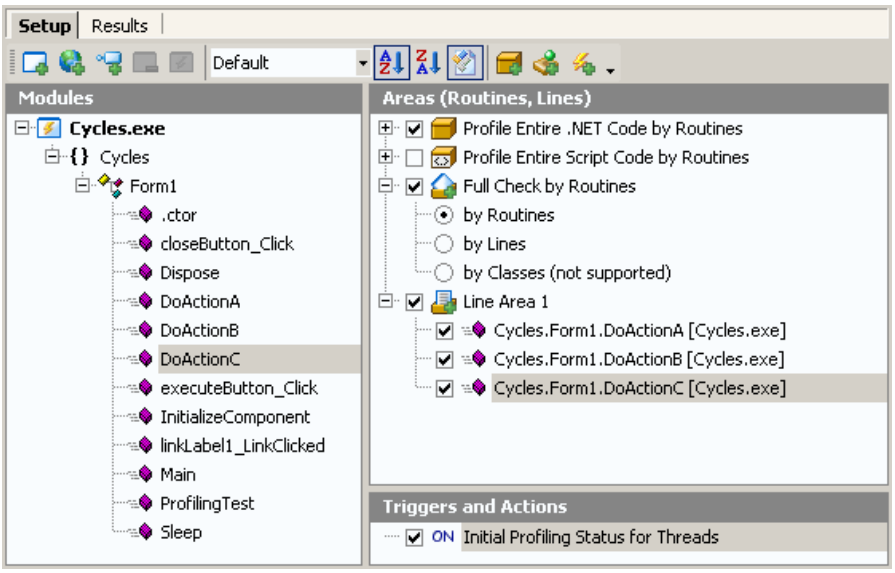
Areas are collections of **elements** to profile. Elements may be source files, classes or single routines. An element may be lodged in more than one area. The reason for having several areas is that an area can be turned on or off by checking or unchecking it. Each area represents, if you wish, a "**typical profiling interest**" for you. Each profile run can take in one or several areas. (If it takes in none, it profiles nothing, unless it's using one of the entire-application profilers listed above.)

By default, an area is an **including** area - it adds elements to the profiling list. However, sometimes you may want to include almost all the methods from a class, or might wish to include a namespace, but skip two routines in it. You can always do this by adding the wanted elements one by one to an (including) area. But, for convenience, you also have the option of adding the entire class or namespace to an Including area, and then defining a special **excluding** area to prevent the profiling of the few sub-elements you wish to skip.

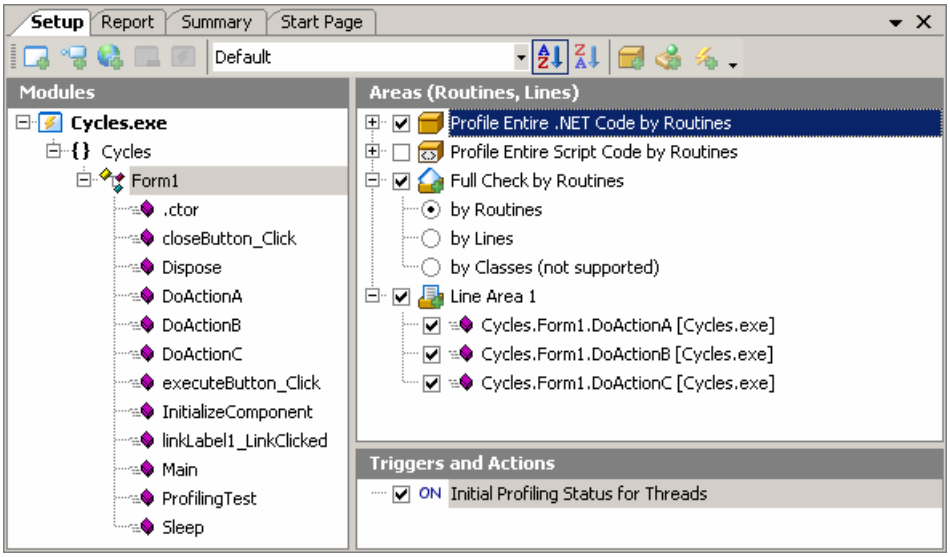
Code specified in the Including areas can be profiled at three levels of detail: class, routine and line. Profiling areas of the **class level** are supported by the Allocation profiler only. These areas specify what classes the profiler will track. Routine and line level areas are supported by the Performance and Coverage profilers. **Routine level** means that AQtime gathers information per routine: how many times it was called, how long it worked, etc. **Line level** means profiling of source code lines within individual routines, classes or namespaces included in the area (see *Profiling Levels*). Class level areas can hold classes, files or modules. Routine and line level areas can hold these elements as well as routines.

An area by itself does not define what you will profile, but what you might wish to profile. What will actually get profiled in a given run, barring other restrictions, is only the checked elements within the checked areas, barring their also being checked in an excluding area. Checking is the object of another topic, *Checking Elements to Profile*. The object of this one is how to build up and maintain the collections of checkable elements, that is, areas themselves.

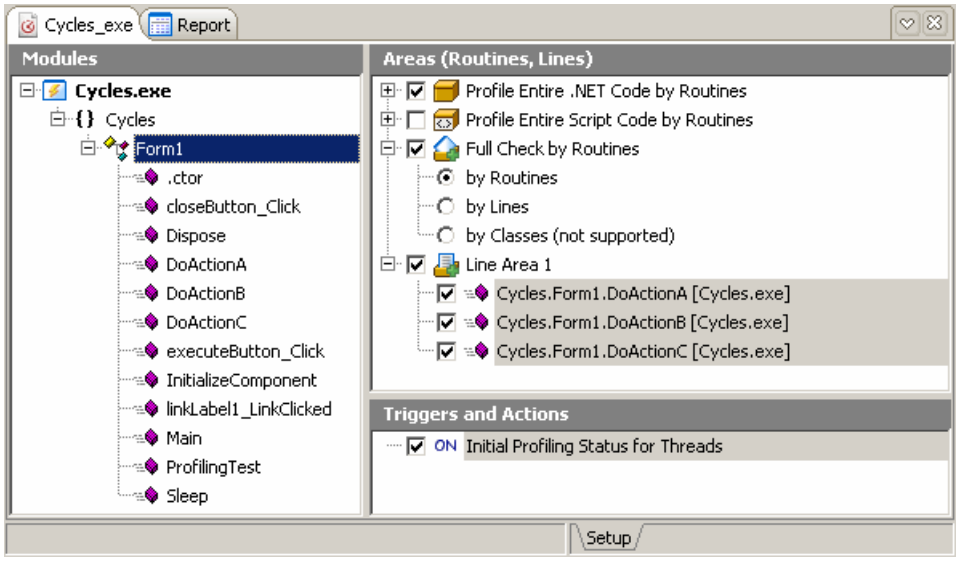
© 2010 AutomatedQA Corp.



Setup panel in AQtime standalone



Setup panel in AQtime integrated into Visual Studio



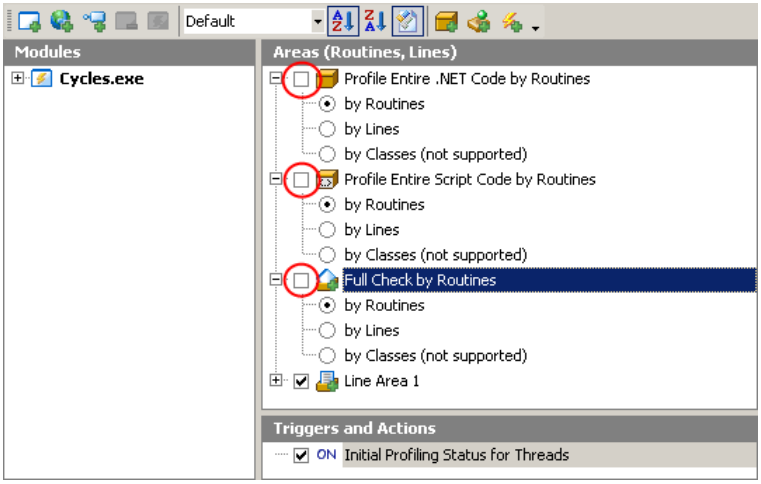
Setup panel in AQtime integrated into Borland Developer Studio

The right-hand pane of the Setup panel holds two lists, **Areas** and **Triggers and Actions**, with areas on top. The separator between the two is moveable; make sure you do see the areas list. Here is how you manage it:

- When you begin a new project, there are three options:
  - ***Full Check*** - If this is checked, AQtime will profile all routines in all modules of the current project.
  - ***Profile Entire Script Code*** - If this is checked, AQtime will profile all routines executed by Microsoft Script Engine. That is, you can profile scripts launched by TestComplete, web browser and other applications that use this engine.
  - ***Profile Entire .NET Code*** - This is similar to *Full Check*. The key difference is that if this box is checked, AQtime will profile all managed modules in the current project as well as all other .NET assemblies that these modules use. This gives you a possibility, for example, to profile functions from the .NET Framework libraries. The *Profile Entire .NET Code* check box overrides all settings that exclude code from profiling (these settings are explained in Excluding Code From Profiling). We would like to note once again that the *Profile Entire .NET Code* setting only affects the managed code. It has no effect on unmanaged code.

  Like other profiling areas, *Full Check*, *Profile Entire Script Code* and *Profile Entire .NET Code* let you specify the level at which the code will be profiled: routine, line or class. For instance, if you select *Full Check by Lines*, AQtime will profile all routines in all modules of the current project at line level. Class level is supported by the Allocation profiler only. Routine and line levels are supported by Performance and Coverage. For more information, see *Profiling Levels*.

To disable the *Full Check* or *Profile Entire .NET Code* option, simply uncheck the appropriate check boxes:



Though profiling all routines is quite convenient, you may not want to include all the application functions in profiling tasks, for instance, to make the profiler run faster. Profiling a large amount of code at line level (for example, when *Full Check by Lines* is selected) can significantly slow down the profiling speed. Therefore, we recommend that you first profile large applications at routine level (for instance, using *Full Check by Routines*). Once you have found problem classes and routines you can add them to a line-level profiling area. That is, profiling areas let you narrow down the places of interest in the application.

- For any normal area, first you need to add it to the list. (It will start out empty.) Click on **Add area** in the context menu and give the area a name in the edit box provided. By default, this will be an including area - So you can specify its level (class, routine or line).
- You can click the **Excluding** radio button in the dialog to make the area excluding. Once you close the dialog, the new area will appear in the Areas list and it will be marked with the appropriate icon:

  - Including class level area
  - Excluding class level area
  - Including routine level area
  - Excluding routine level area
  - Including line level area

  Areas that are not supported by the current profiler have grayed icons, but you still can add elements to them (see below).

- Add elements to the area. The easiest way to do this is to go to the left-hand panel. This displays a treeview of the entire application, any part of which you can expand or contract. You can make single or multi-selections in the view, using click, Shift-and-drag or Ctrl-click. You can for instance select an entire namespace, then unselect parts of it by Ctrl-clicking them off. To add the selected elements to the area you want, drag them from the tree-view and drop onto the area in the right-hand pane, or right-click any of the selected elements and choose **Add Selected to Area | <*Area name*>** from the context menu.

  If you add a module or a class to an area, all routines in this module or class will be profiled at the appropriate level.

  Profiling under Visual Studio or Borland Developer Studio provides a possibility to add elements to the area directly from the Code Editor. AQtime adds the **Profile** submenu to the editor's context menu. The **Profile** submenu has the following items:

  - **Profile <routine name> Routine** - Starts profiling a routine under the cursor.

- **Profile <class name> Class** - Starts profiling a class where the cursor is located in the code.
- **Profile <file name> File** - Starts profiling a source file displayed in the code editor.

When any of those items are chosen, AQtime creates or redefines the line area named **Quick profiling area** and adds the chosen element to the profiling list and immediately runs the selected profiler.

Notes:

- Routines that belong to triggers or actions are treated as rotuines that are added to a profiling area. They are profiled even if they belong to an excluding area. For more information, see *Using Actions* and *Setting Up Triggers*.
- The Setup panel does not display classes that neither introduce new methods, not override existing ones. This may happen even if a class introduces a new method, but this method is not called in the application code: the compiler may not include the method in the application's binary code during optimization. However, profiling of such classes is possible if you enable the *Full Check* option. AQtime will profile the methods of the ancestor class. For more information on profiling such classes with the Allocation profiler, see a note in the Allocation Profiler topic.
- Note for Borland Developer Studio users: when you start profiling in BDS with the **Run with profiling** command, AQtime disables any custom areas and profiles the application with the *Full Check by Routines* and *Profile Entire .NET Code by Routines* options. To profile within custom areas use one of the following ways to start profiling:
  - Select **Run** menu item or click **Run** button on the **Debug** toolbar or press F9, while the AQtime project is active in Borland Developer Studio's **Project Manager**.
  - Right-click the AQtime project in the **Project Manager** and select **Run** from the context menu.

You can add elements at any time to any existing area. Elements can also be removed at any time. They can be dragged away from the area, and dropped onto the left-hand pane. More conveniently, you can uncheck the elements you wish to remove, then use **Remove Unchecked Elements** from the context menu. Alternatively, you can check only the elements to remove and use **Remove Checked Elements**.

AQtime stores the profiling area settings in the project file (**.aqt**).

Remember that you do not have to use an area as-is. The point of adding or removing elements is to create a "stored definition", which you can later trim simply by unchecking elements. It is perfectly reasonable to run a profiler on an entire area, then begin unchecking elements for each successive run, as you eliminate the uninteresting parts.

Make sure that when you profile a routine, part of its execution time will be spent calling other routines, which we call "child" routines (it's the calls that are child calls, actually). Unless the child routines are part of the profiling area, you will not be able to narrow down your profiling to find bottlenecks outside the main routine. Profiling results can discriminate between a routine's own execution time and its overall execution time, "with children". But if you want to know about the children themselves, they have to be added to the profiling area. See *Profiling Child Routines Along With Parents*.

Checking Elements to Profile

Areas are collections of elements (namespaces, classes or routines) which you keep together because you might want to profile them. In other words, the Areas list is a repository of profileable elements grouped into collections called areas, and an element may belong to more than one collection.

Normally, as you start profiling an application, you use profilers over broad areas. Once you know what you want needs to be tracked, you may run profilers over smaller areas (to make the profiling faster). In an ordinary profile test, the pattern is that, from run to run, you set areas so as to profile less and less.

On any run, only the areas you have checked will be profiled. If no area is checked, no profiling will occur. Often times however, you will want to trim down within an area. Open any area in the tree view, and you see the elements you put in it. You can temporarily remove any element from profiling by unchecking it individually.

You can also use the common multi-select commands (Shift-drag and Ctrl-click) to select a larger number of elements at once, then use the context menu's **Check Selected** or **Uncheck Selected** commands.

Once you begin a new profiling session, remember to open the areas you have checked, so that you can see what elements are currently checked or unchecked within them. If you check an area, only the currently checked elements in it will be profiled. If an area is not checked, then none of its elements appear checked. Checked elements only show up once the area has been checked.

All of the above is meant for the normal, including, areas. The elements of excluding areas only become excluded if their area is checked, and they themselves are checked. Normally, you will simply check any excluding area when you check the including area for which it holds exceptions.

Note for Borland Developer Studio users: when you start profiling in BDS with the **Run with profiling** command, AQtime ignores the profiling areas and profiles the application with the default *Full Check by Routines* and *Profile Entire .NET Code by Routines* settings. To profile with areas, use one of the following to start profiling:

- Select **Run** menu item or click **Run** button on the **Debug** toolbar or press F9, while the AQtime project is active in Borland Developer Studio's **Project Manager**.
- Right-click the AQtime project in the **Project Manager** and select **Run** from the context menu.

Using Triggers

Triggers are an AQtime facility allowing better control of profiling under the Performance, Coverage and Function Trace profilers. "Triggers" include three linked settings: *on-triggers*, *off-triggers* and *Initial profiling status*.

AQtime offers several means of **restricting** what parts of an application get profiled. Triggers provide a crucial means to fine-tune exclusions, but they work in association with the other means. In the end, what gets profiled is what currently falls under no restriction of one kind or another. See *Controlling What to Profile* for full details.

Especially, since areas and triggers are rather similar, it is important to understand that, no matter what triggers and Initial Status may decree, nothing will get profiled if it is not checked in the areas section.

The purpose of triggers is to allow profiling to come on when certain routines (on-triggers) are executing (including during their calls to "child" routines), or on the contrary to be turned off (off-triggers). With areas and checking, you set what you **never** want profiled during the current run (that is, everything but the checked elements). With triggers and Initial Status, you set in **what part of the execution path** you want to enable profiling, or to disable it.

Technically, what triggers do is very simple to explain:

- For any given thread, at any given moment, **profiling status** is "enabled" or "disabled". Profiling actually operates in a thread if, one, status for that thread is "enabled" and, two, nothing else excludes the current routine from profiling.
- The **Enable/Disable Profiling** button (**the Profile | Enable/Disable Profiling** menu item in Visual Studio or the **Enable/Disable Profiling** button on Borland Developer Studio's

Run.AQtime toolbar) is an onscreen way of controlling profiling status by hand while the application runs. If it is pressed-in ( ), profiling status is as set otherwise. When you unclick it ( ), profiling is turned off for all threads. When you press it back in, you re-enable profiling, according to whatever is otherwise set for the application at that point (not as it was when you disabled it).

- If there are triggers of any kind, the **Initial Profiling Status** option specifies whether the profiling on or off on the creation of each new thread. In other words, Initial Profiling Status defines whether profiling is enabled or not before any trigger in a thread starts executing. After that, it has no effect. Likewise if there are no triggers at all.
- When an **on-trigger** routine begins executing, it saves the current profiling status for its thread and enables profiling for that thread. This still applies to everything it calls. When it reaches the end of its execution (after perhaps hundreds of calls, sub-calls and sub-sub-calls), it restores the thread's profiling status as it found it.
- When an **off-trigger** routine begins executing, it saves the current profiling status for its thread and turns profiling off for that thread. When it reaches the end of its execution, it restores the thread's profiling status as it found it.

Suppose, `Proc_B` is an off-trigger routine, profiling is currently enabled and either *Full Check by Routines*, or *Full Check by Lines* is used:

```
Proc_A;
Proc_B // off-trigger routine
  Proc_D; // Proc_D and Proc_E are child routines of Proc_B,
  Proc_E; // that is, they are called within Proc_B.
  // Proc_D and Proc_E are not profiled.
Proc_C;
```

As profiling is enabled and *Full Check* is on, AQtime profiles `Proc_A`. When the application enters `Proc_B`, profiling is disabled for that thread. So `Proc_D` and `Proc_E` are not profiled. When `Proc_B` exits, AQtime restores the profiling status as it was - enabled -- so `Proc_C` is profiled.

This topic has explained the **use** of triggers. It has not explained **options** for triggers. Nor has it explained **how to set them up**. Both questions are the object of the next topic.

Note that besides triggers AQtime offers another facility to enable and disable the profiling status during the run - *actions*. They are similar to triggers and will be explained in another topic.

Setting Up Triggers

Before setting up triggers, make sure you have read *Using Triggers*. A few reminders:

- Triggers work in conjunction with the other means of selecting what to profile. Execution is actually profiled only when all means say "yes". See *Controlling What to Profile*.
- Of all means of selecting what to profile, triggers are the only one that can select on a thread basis.
- "Triggers" include three linked settings: *on-triggers*, *off-triggers* and *Initial profiling status*.
- Triggers apply to the Performance and Coverage profilers only.

Triggers are defined and controlled in the Setup panel. The right-hand pane holds two lists, **Areas** and **Triggers and Actions**, with *Triggers and Actions* at the bottom. The separator between the two is moveable; make sure you see the triggers list.

In that list, a "trigger" is a collection of *potential* trigger routines of one type, either on- or off-. No triggers operate at all unless at least one collection is **checked**. And in the collection only those triggers operate which are checked also.

Up to this point, the mechanics of managing triggers, including checking them on or off, is identical to that of managing areas. We refer you to *Defining Areas to Profile* and *Checking Elements to Profile* for details.

Now for the points that are **different** with setting up triggers:

- If for instance, one element of a trigger is a namespace and you check it, then each single routine in that namespace becomes a trigger. Uncheck it and no routine in the namespace acts as a trigger, unless it also belongs to another checked collection, and it is checked in there. In other words, you should be more conservative when including elements in triggers than when including them in areas. After a certain point, more triggers simply mean more confusion, when the whole purpose of triggers is to clarify the profiling results.
- A "trigger" collection is defined as holding **on-** or **off-** triggers (one type per collection), not just triggers.
- AQtime treats routines added to a trigger as routines added to a profiling area. This is not dependent on the trigger type (on or off). The profiling level depends on whether the routine belongs to a profiling area:
  - If the routine is not added to any profiling area, it will be profiled at routine level.
  - If the routine belongs to a line-level or routine-level area, it will be profiled at line level or routine level correspondingly.
  - If the routine belongs to an excluded area, it will still be profiled at routine level. That is, trigger routines cannot be excluded from profiling.
- There is one default trigger always at the top of the list - **Initial Profiling Status for Threads**. It sets the profiling status at the beginning of threads, before any other triggers click in (after that, it has no effect). The *Initial profiling status* can be set on (profiling allowed until an off-trigger operates) or off (no profiling until an on-trigger operates).
- Except for the Initial Status trigger, each trigger can be tuned regarding **after how many calls** a trigger will start operating, and then **for how many calls** (after which it will become inoperative). These two options are themselves affected by another option - is the call count taken over all threads, or over each thread individually? This yes-no option, which sets the meaning of the ones explained below, is **For All Threads**. Remember that, whether or not calls are counted over all threads, triggers only act on their own thread.
- The **Pass Count** option allows the trigger routine or routines to be called a certain number of times before they act as triggers. For instance, this lets you skip the startup phase of your application. Pass Count 3 means "only act as trigger on the fourth call". By default Pass Count is 0 and operates from the first call.
- The **Work Count** option then sets how many consecutive calls will act on the trigger (to allow profiling or to disallow it), before the trigger stops acting again. This is a way to limit the amount of data gathered at the point where it just repeats what's known. 0 is again the default, but here it does not mean "never activate", it means "stay activated to end of run".
- Finally, the **Cycling** yes-no option (default no) sets whether the pass-count-work-count cycle will repeat after work count + 1 is reached and the trigger is deactivated. A cycling trigger is a way to sample application behavior through various phases without amassing data for every single call.

AQtime stores the trigger settings in the project file (**.aqt**).

Note for Borland Developer Studio users: when you start profiling in BDS with the **Run with profiling** command, AQtime ignores triggers and profiles the application with the default settings. To profile with triggers, use one of the following to start profiling:

- Select **Run** menu item or click **Run** button on the **Debug** toolbar or press F9, while the AQtime project is active in Borland Developer Studio's **Project Manager**.
- Right-click the AQtime project in the **Project Manager** and select **Run** from the context menu.

Using Actions

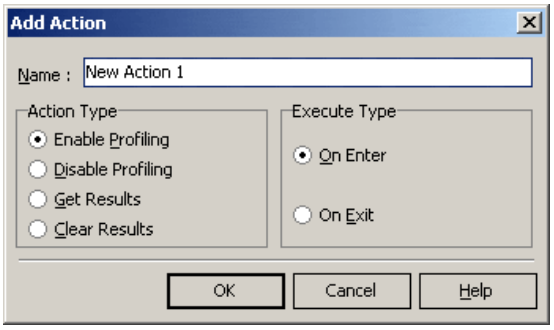
An **action** is a routine, at the beginning or end of which AQtime performs specific actions: switching the profiling status on or off or getting profiling results. Actions are very similar to triggers in the sense that they allow you to switch the profiling status during the profiler run. However, actions function a bit differently than triggers:

- First of all, actions do not save and restore profiling status for the thread. That is, actions do not restore the profiling status for the thread after the action routine is over. When the trigger routine is over, AQtime restores the profiling status for the thread where the trigger executed.
- Unlike triggers, which change the profiling status at the beginning of a trigger routine, actions can run either at the beginning or at the end of a routine.

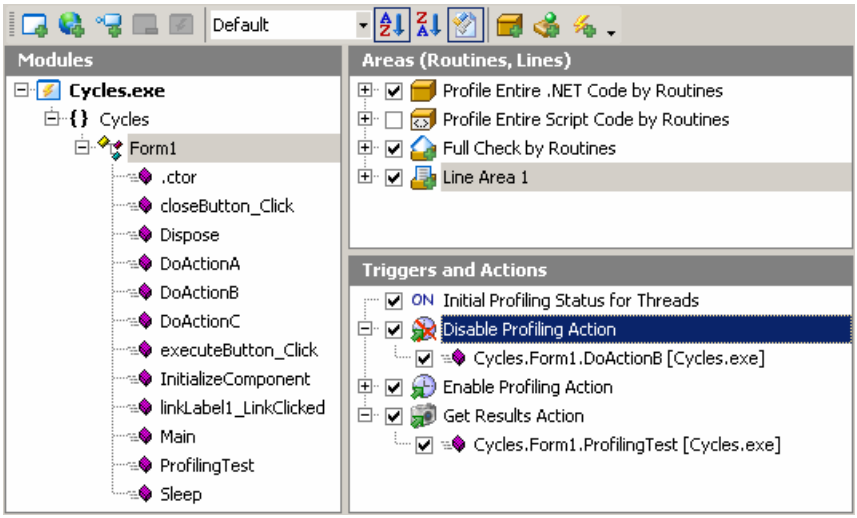
One of the features that make actions great is the ability to command AQtime to get profiling results during the profiling run. Without actions, you can obtain profiling results only by selecting **Run | Get Results** from AQtime's main menu (**Profile | Get Results** in Visual Studio's menu or **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar) or when the profiled process is over. Since, two profiler runs always differ from each other, actions provides automatic result generation at the exact moments when it is needed.

Currently, actions are only supported by the Performance, Coverage and Function Trace profilers.

The list of existing actions is displayed in the **Triggers and Actions** section of the Setup panel. To create a new action, right-click somewhere in this section, select **Add Action** from the context menu and specify the action name and type in the subsequent Action Properties dialog:



The newly created action is displayed in the Triggers and Actions list of the Setup panel:



The action icon indicates the action type:

Actions executed upon entering the routine	Actions executed upon exiting the routine
- “Enable Profiling” action	- “Enable Profiling” action
- “Disable Profiling” action	- “Disable Profiling” action
- “Get Results” action	- “Get Results” action
- "Clear Results" action	- "Clear Results" action

To check or change properties of an existing action, right-click this action in the Setup panel and select **Edit** from the context menu. This will call the Action Properties where you can modify action properties.

Once you have created an action, you can add routines, classes, namespace and modules to it. You can do this in the same manner, as you add elements to profiling areas and triggers: by dragging the desired elements from the treeview on the right of the Setup panel to the desired action. See also *Defining Areas to Profile* and *Setting Up Triggers*. You can also select the desired element in the treeview (you can use Shift and Ctrl for multiselection) and then choose **Add Selected to Action | <*action_name*>** from the context menu item.

Notes:

- If you add a class, namespace or module to an action, then every single routine in this class, namespace or module will function as an action.
- Adding a routine to an action is analogue to adding a routine to a profiling area and profiling it with trigger settings that correspond to the action type. Below are two examples:
  - Suppose that you have added a routine to the action of the Enable Profiling type (analogue to an on-trigger), the action is performed upon entering the routine and the routine is added to a routine-level area. In this case, AQtime will profile the routine at routine level and you will see the routine results in the Report panel.

    If the routine was added to a line-level area, AQtime would profile it at line level. If the routine was not added to any area, AQtime would profile it at routine level.

    Furthermore, if the action has the described settings, AQtime will profile the routine even if it is added to an excluding profiling area.
  - In the second scenerio, suppose that you have added a routine to the action of the Disable Profiling type (analogue to an off-trigger), and the action is performed upon entering the routine. Regardless of whether the routine is added to any profiling area, the routine will not be profiled, since AQtime disables profiling when the routine starts executing.

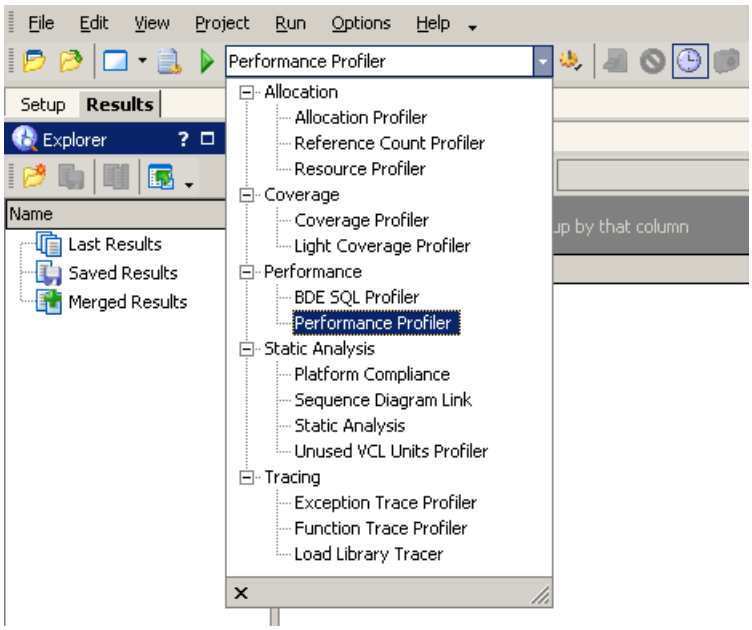
In other words, adding a routine to an action ignores the “area” settings for the routine, but does not ignore the “trigger” settings.

- Note for Borland Developer Studio users: when you start profiling in BDS with the **Run with profiling** command, AQtime ignores actions. To profile with actions, use one of the following to start profiling:
  - Select **Run** menu item or click **Run** button on the **Debug** toolbar or press F9, while the AQtime project is active in Borland Developer Studio’s **Project Manager**.
  - Right-click the AQtime project in the **Project Manager** and select **Run** from the context menu.

Like area and trigger settings, the action settings are stored in the AQtime project file (**.aqt**).

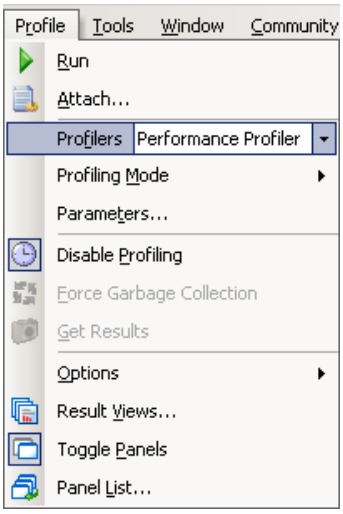
Selecting a Profiler

One AQtime “run” is one execution of the application under one profiler. If you use AQtime standalone, you can select the profiler to be run from the dropdown list on the Standard toolbar, just to the right of the **Run** button:

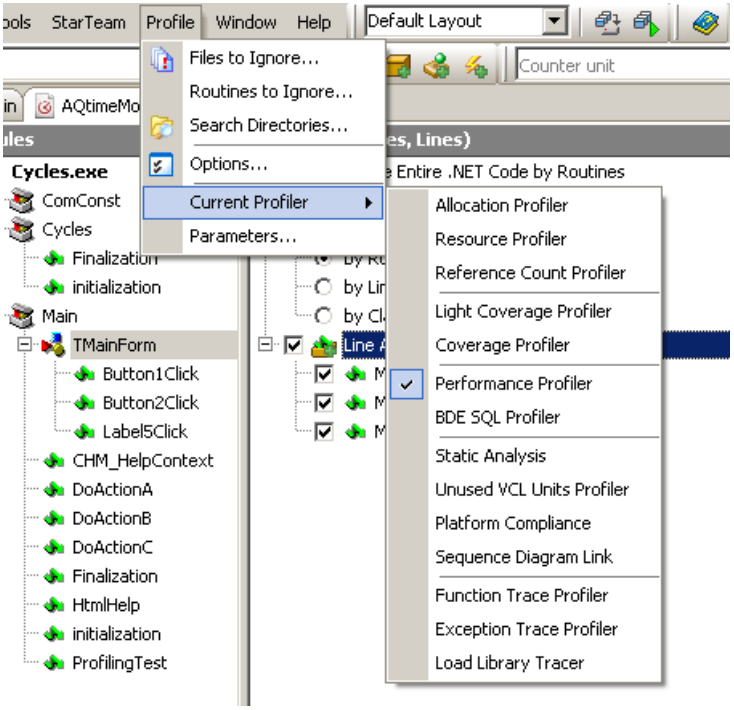


The dropdown list is actually a treeview. Individual profilers are listed when you open a branch.

If you use AQtime integrated into Visual Studio, you can select the profiler to be run from the **Profiler** dropdown list in the **Profile** menu:



If you use AQtime integrated into Borland Developer Studio, you can select the profiler to be run from the **Profiler** dropdown list in the **Profile** menu:



AQtime stores the current profiler selection in the project file (**.aqt**) when you close the project. Next time you open your project in AQtime, the latter will automatically select the profiler that was active when AQtime was closed.

Profiling and Analyzing Results

Doing One Profiler Run

Before doing your first run of your application under an AQtime profiler, you must have completed these preliminary steps:

- *Compiling your application with or without debug information*
- *Opening the project (application) in AQtime*
- *Controlling what to profile*
- *Selecting the profiler to run*

Between each run, you are likely to change what to profile, and fairly often to change profilers as well. Additionally, during the life of the project, you will of course frequently recompile your application after making changes.

Note: We would like to note that if you use a computer that has several processors or a multiple-core processor (for example, dual-core CPU) and has Windows XP Service Pack 2, then you **must install the Windows update #896256** in order for AQtime to be able to profile your application correctly. The update is available on Microsoft's web site:

- http://support.microsoft.com/kb/896256

Now, with the four preliminary steps completed, there are a few more checks to go through before starting the run:

- Check that you have set the running conditions as they need to be. For an EXE, these are the (possible) runtime arguments, for a DLL, the (necessary) host application. Most of the time, you do not have to do anything, because you are testing an exe that takes no parameters, or because you simply want to keep the existing settings. To check or change conditions, use **Run | Parameters** from AQtime's main menu, **Profile | Parameters** from Visual Studio's menu or the **Parameters** button on Borland Developer Studio's **Run.AQtime** toolbar. This leads you to the **Run Parameters** dialog, which has a box both for parameters and for host application. See *Profiling Dynamic Link Libraries* for details on the latter.
- Make sure that the necessary modules (for example, DLLs) can be loaded.
- Specify the type of profiled executable using the **Profiling Mode** dropdown list box on AQtime's **Standard** toolbar (Visual Studio's **AQtime** toolbar or Borland Developer Studio's **Run.AQtime** toolbar). This list includes the following items:

Normal	Means that the profiled executable is a regular managed or unmanaged executable or library. This item is selected by default.
ASP.NET	Means that the profiled executable is an ASP.NET application or .NET Web service. To profile an NT service, select Service profiling. For more information on how to profile ASP.NET applications, see *Profiling ASP.NET Applications*.
IIS	Means that the profiled executable is an IIS application or Web service created with an unmanaged compiler. For more information on profiling such applications, see *Profiling IIS Applications*.
COM Server	Means the profiled executable is a COM application. For more information, see *Profiling COM Applications*.

Service Means that the profiled executable is an NT service. Don't use it for ASP.NET service profiling. For more information on how to profile services, see *Profiling Services*.

- Make sure that the **Enable/Disable Profiling** button on the Standard toolbar is in its normal pressed-in state (as shown), unless you want to start with profiling turned off, overriding your trigger and action settings.
- If you are using the Performance or Function Trace profiler and your computer's CPU (for instance, Intel Pentium M) supports dynamic CPU frequency mode, you should disable the dynamic change of the CPU frequency in order to obtain accurate results. If the CPU frequency is changed dynamically, the timing results may be inaccurate.
- If you are using the Allocation profiler, make sure that you have checked one or more class-level profiling areas. Otherwise, you may receive empty results. This typically concerns profiling of .NET applications and the reason for this is quite simple: the profiler *always* tracks memory-block allocations done by non-class memory management routines such as `new` or `alloc`. Therefore, if you start profiling a .NET application and there are no class-level areas selected, the profiler will not notify you, since that .NET application may include unmanaged sections of code and these sections may call non-class memory management routines, which you may want to profile.

Note: In order for AQtime to be able to profile your application, the user account, under which AQtime will be running, must have administrator permissions. The easiest way to grant these permissions is to add this account to the Administrators group.

If you use AQtime standalone, to start profiling simply press **Run** on the Standard toolbar or select **Run | Run** from AQtime's main menu.

If you use AQtime integrated into Visual Studio, select **Profile | Run** from Visual Studio's main menu. An alternative way to start profiling is to press Visual Studio's **Run** button or selecting **Debug | Run** menu item while one of AQtime's panels is active or while an AQtime panel is selected in the Solution Explorer.

If you use AQtime integrated into Borland Developer Studio, select **Run | Run With Profiling** from Borland Developer Studio's main menu. An alternative way to start profiling is to press Borland Developer Studio's **Run** button on the Debug toolbar or select the **Run | Run** menu item while one of AQtime's panels is active or while an AQtime panel is selected in the Project Manager.

The difference between the ways to start profiling consists in how custom profiling areas will be handled. Using the **Run With Profiling** command resets custom areas (if any) and profiles with the default *Full Check* and *Profile Entire .NET Code* options. Whereas to start profiling via the **Run** command preserves custom areas.

After you selected Run, AQtime displays the Run Settings dialog where you can check or modify profiling options and conditions for the coming profiler run. AQtime will start profiling after you close the dialog.

x64 editions of the Windows operating system require that the "bitness" of a process and the module that is loaded into this process be the same (in other words, 32-bit modules can be loaded only into a 32-bit process and 64-bit modules can be loaded only into a 64-bit process). This limitation may exist when profiling a dynamic link library, an in-process COM server or any other module that is loaded into a process. If the "bitness" of the module and process is not the same, AQtime will stop profiling and display an error message informing you about the problem.

Some points while executing the application:

- Run the operations that you need to profile (for instance, those where you suspect a bottleneck). You might **plan your run** before starting, to be sure you hit all the high (or rather, low) points.

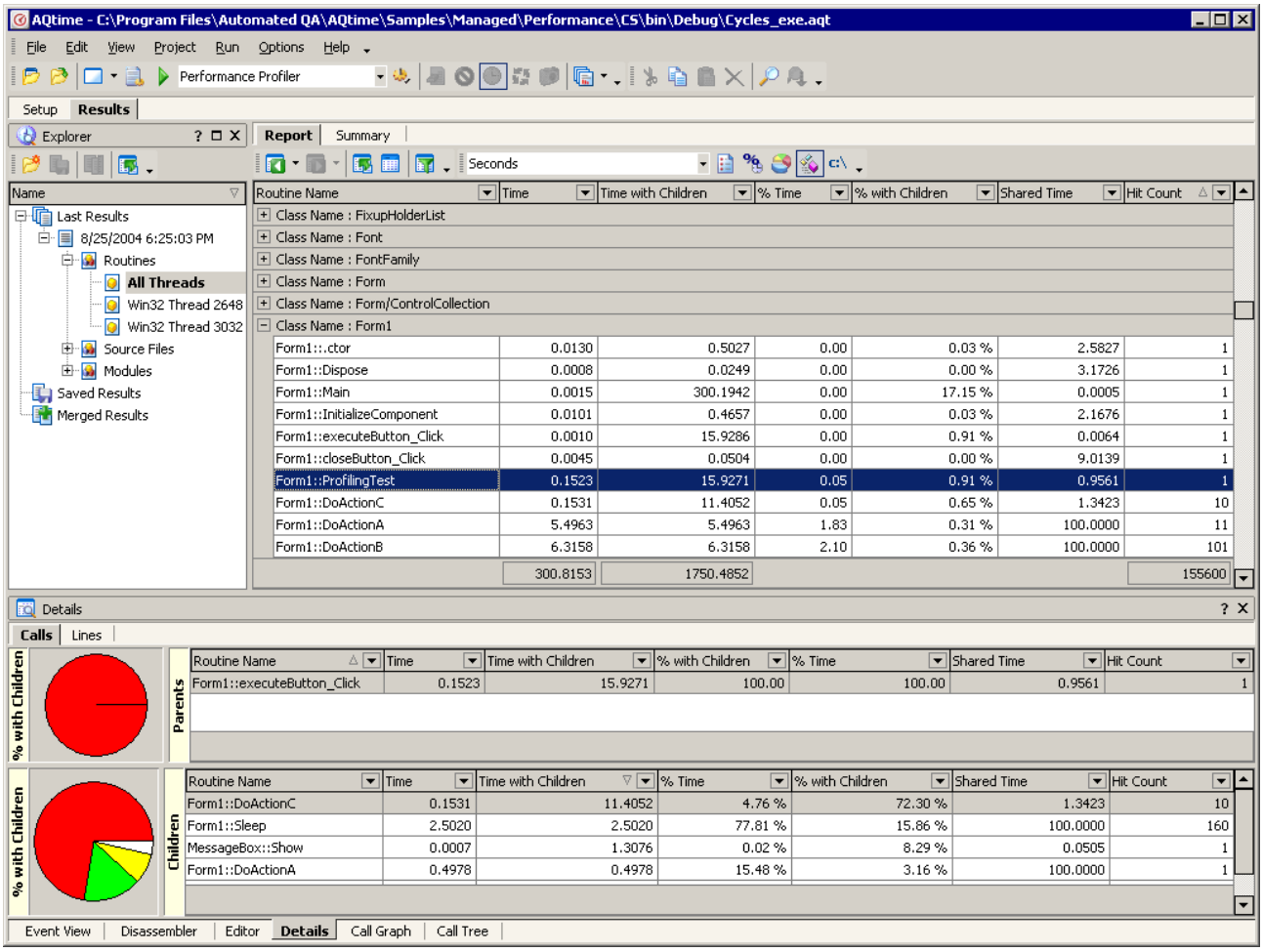
- You can use the **Enable/Disable Profiling** button (the **Profile | Enable/Disable Profiling** menu item in Visual Studio or the **Enable/Disable Profiling** button on the **Run.AQtime** toolbar in Borland Developer Studio) to suspend profiling, but not execution, while you run through parts of the application you do not need to profile. See *Controlling What to Profile* for the caveat.
- AQtime generates results when the application execution is over. To obtain profiling results during the run, select **Get Results** from the **Run** menu or from the **Standard** toolbar (if you use AQtime integrated into Visual Studio, select **Profile | Get Results** from Visual Studio's main menu; if you use AQtime integrated into Borland Developer Studio, click **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar). You may also need to use this item if the profiling process never ends. See *Getting Results During Testing* for more information.
- If you profile a managed application, you can click **Force Garbage Collection** to initiate garbage collecting in the profiled process (if you used AQtime integrated into Visual Studio, select **Profile | Force Garbage Collection** from Visual Studio's main menu; if you use AQtime integrated in Borland Developer Studio, click **Force Garbage Collection** on the **Run.AQtime** toolbar). You may do this, for instance, to see which objects will be removed and which will remain in memory.
- If you want to force the application process to end, click **Terminate** (if you use AQtime integrated into Visual Studio, select the **Profile | Terminate** menu item; if you use AQtime integrated into Borland Developer Studio, click **Terminate** on the **Run.AQtime** toolbar). You may need to do this if the application cannot normally be ended without rebooting, logging off or some other drastic intervention, or if it simply stopped responding. Pressing Terminate does not generate profiling results.
- Some profilers, with some over-enthusiastic settings, may slow application execution to the point where you mistake it for a crash.
- Once you have gone through the operations you wanted, exit the application. Do not accumulate needless profile data.

If this has happened, you can flush all gathered results. To do this, select **Clear Results** from the **Run** menu, or from the **Standard** toolbar (or select **Profile | Clear Results** from Visual Studio's main menu, or click **Clear Results** on Borland Developer Studio's **Run.AQtime** toolbar). See *Clearing Results During Profiling*.

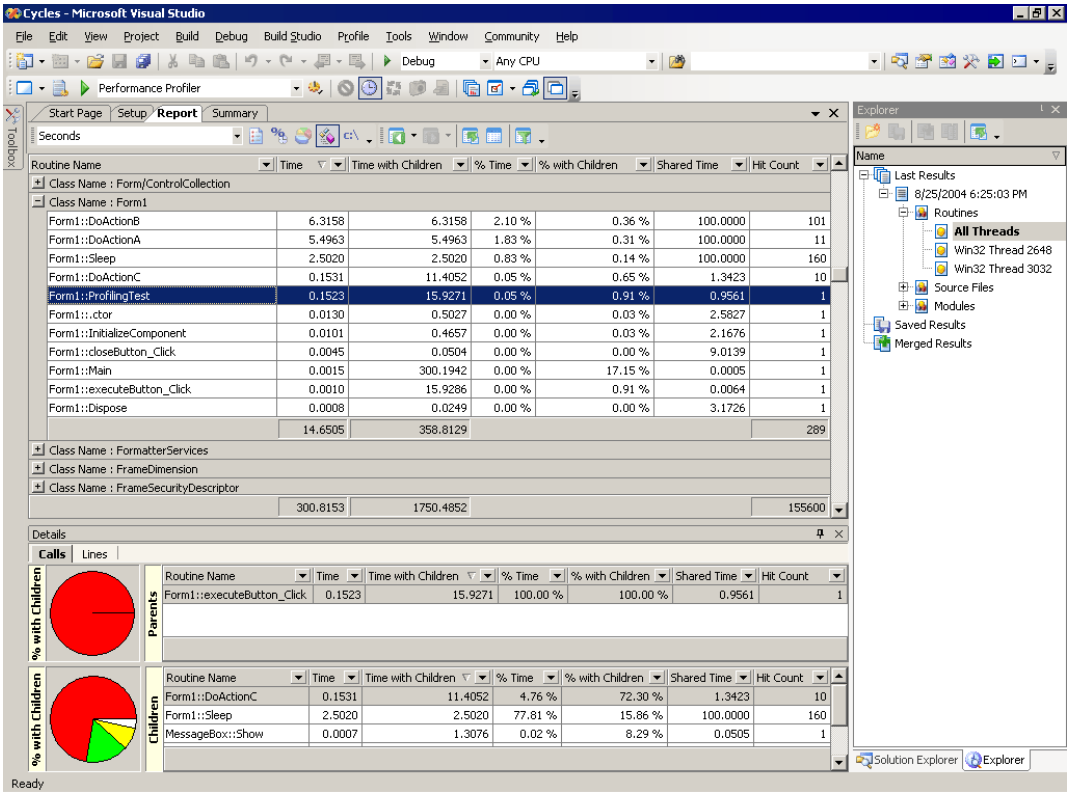
Once you exit, the resultant profile will be displayed in AQtime's Report panel.

Analyzing Profiler Results

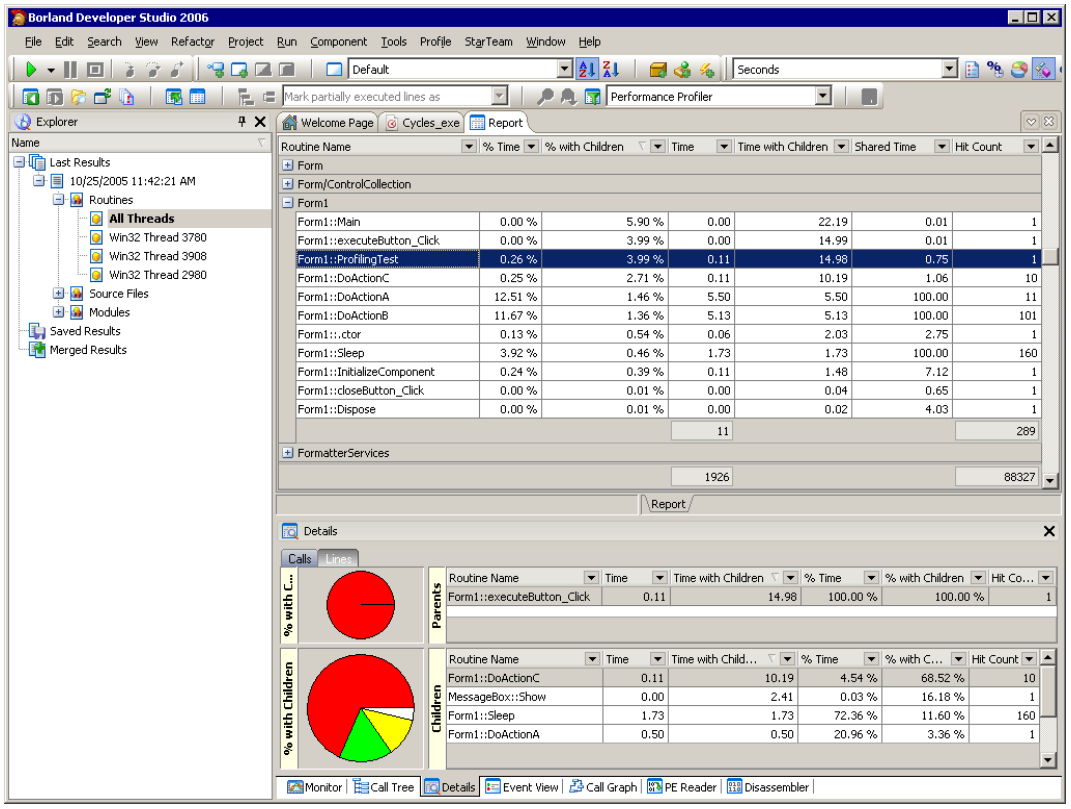
After profiling your application, AQtime displays profiling results in its panels:


AQtime - C:\Program Files\Automated QA\AQtime\Samples\Managed\Performance\CS\bin\Debug\Cycles_exe.aqt
File Edit View Project Run Options Help
Performance Profiler
Setup Results
Explorer
Report Summary
Seconds
Name
Last Results
8/25/2004 6:25:03 PM
Routines
All Threads
Win32 Thread 2648
Win32 Thread 3032
Source Files
Modules
Saved Results
Merged Results
Routine Name Time Time with Children % Time % with Children Shared Time Hit Count
Class Name : FixupHolderList
Class Name : Font
Class Name : FontFamily
Class Name : Form
Class Name : Form/ControlCollection
Class Name : Form1
Form1::.ctor 0.0130 0.5027 0.00 0.03 % 2.5827 1
Form1::Dispose 0.0008 0.0249 0.00 0.00 % 3.1726 1
Form1::Main 0.0015 300.1942 0.00 17.15 % 0.0005 1
Form1::InitializeComponent 0.0101 0.4657 0.00 0.03 % 2.1676 1
Form1::executeButton_Click 0.0010 15.9286 0.00 0.91 % 0.0064 1
Form1::closeButton_Click 0.0045 0.0504 0.00 0.00 % 9.0139 1
Form1::ProfilingTest 0.1523 15.9271 0.05 0.91 % 0.9561 1
Form1::DoActionC 0.1531 11.4052 0.05 0.65 % 1.3423 10
Form1::DoActionA 5.4963 5.4963 1.83 0.31 % 100.0000 11
Form1::DoActionB 6.3158 6.3158 2.10 0.36 % 100.0000 101
300.8153 1750.4852 155600
Details
Calls Lines
% with Children
Parents
Routine Name Time Time with Children % with Children % Time Shared Time Hit Count
Form1::executeButton_Click 0.1523 15.9271 100.00 100.00 0.9561 1
% with Children
Children
Routine Name Time Time with Children % Time % with Children Shared Time Hit Count
Form1::DoActionC 0.1531 11.4052 4.76 % 72.30 % 1.3423 10
Form1::Sleep 2.5020 2.5020 77.81 % 15.86 % 100.0000 160
MessageBox::Show 0.0007 1.3076 0.02 % 8.29 % 0.0505 1
Form1::DoActionA 0.4978 0.4978 15.48 % 3.16 % 100.0000 1
Event View Disassembler Editor Details Call Graph Call Tree

Report panel in AQtime standalone


Cycles - Microsoft Visual Studio
File Edit View Project Build Debug Build Studio Profile Tools Window Community Help
Debug
Any CPU
Performance Profiler
Start Page Setup Report Summary
Seconds
Routine Name Time Time with Children % Time % with Children Shared Time Hit Count
Class Name : Form/ControlCollection
Class Name : Form1
Form1::DoActionB 6.3158 6.3158 2.10 % 0.36 % 100.0000 101
Form1::DoActionA 5.4963 5.4963 1.83 % 0.31 % 100.0000 11
Form1::Sleep 2.5020 2.5020 0.83 % 0.14 % 100.0000 160
Form1::DoActionC 0.1531 11.4052 0.05 % 0.65 % 1.3423 10
Form1::ProfilingTest 0.1523 15.9271 0.05 % 0.91 % 0.9561 1
Form1::.ctor 0.0130 0.5027 0.00 % 0.03 % 2.5827 1
Form1::InitializeComponent 0.0101 0.4657 0.00 % 0.03 % 2.1676 1
Form1::closeButton_Click 0.0045 0.0504 0.00 % 0.00 % 9.0139 1
Form1::Main 0.0015 300.1942 0.00 % 17.15 % 0.0005 1
Form1::executeButton_Click 0.0010 15.9286 0.00 % 0.91 % 0.0064 1
Form1::Dispose 0.0008 0.0249 0.00 % 0.00 % 3.1726 1
14.6505 358.8129 289
Class Name : FormatterServices
Class Name : FrameDimension
Class Name : FrameSecurityDescriptor
300.8153 1750.4852 155600
Details
Calls Lines
% with Children
Parents
Routine Name Time Time with Children % Time % with Children Shared Time Hit Count
Form1::executeButton_Click 0.1523 15.9271 100.00 % 100.00 % 0.9561 1
% with Children
Children
Routine Name Time Time with Children % Time % with Children Shared Time Hit Count
Form1::DoActionC 0.1531 11.4052 4.76 % 72.30 % 1.3423 10
Form1::Sleep 2.5020 2.5020 77.81 % 15.86 % 100.0000 160
MessageBox::Show 0.0007 1.3076 0.02 % 8.29 % 0.0505 1
Explorer
Name
Last Results
8/25/2004 6:25:03 PM
Routines
All Threads
Win32 Thread 2648
Win32 Thread 3032
Source Files
Modules
Saved Results
Merged Results
Solution Explorer Explorer
Ready

Report panel in AQtime integrated into Visual Studio

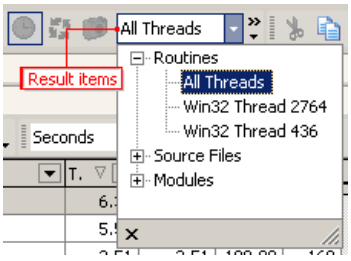

Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Default
Seconds
Mark partially executed lines as
Performance Profiler
Explorer
Name
Last Results
10/25/2005 11:42:21 AM
Routines
All Threads
Win32 Thread 3780
Win32 Thread 3908
Win32 Thread 2980
Source Files
Modules
Saved Results
Merged Results
Welcome Page Cycles_exe Report
Routine Name % Time % with Children Time Time with Children Shared Time Hit Count
Form
Form/ControlCollection
Form1
Form1::Main 0.00 % 5.90 % 0.00 22.19 0.01 1
Form1::executeButton_Click 0.00 % 3.99 % 0.00 14.99 0.01 1
Form1::ProfilingTest 0.26 % 3.99 % 0.11 14.98 0.75 1
Form1::DoActionC 0.25 % 2.71 % 0.11 10.19 1.06 10
Form1::DoActionA 12.51 % 1.46 % 5.50 5.50 100.00 11
Form1::DoActionB 11.67 % 1.36 % 5.13 5.13 100.00 101
Form1::.ctor 0.13 % 0.54 % 0.06 2.03 2.75 1
Form1::Sleep 3.92 % 0.46 % 1.73 1.73 100.00 160
Form1::InitializeComponent 0.24 % 0.39 % 0.11 1.48 7.12 1
Form1::closeButton_Click 0.00 % 0.01 % 0.00 0.04 0.65 1
Form1::Dispose 0.00 % 0.01 % 0.00 0.02 4.03 1
11 289
FormatterServices
1926 88327
Report
Details
Calls Lines
% with C...
Parents
Routine Name Time Time with Children % Time % with Children Hit Co...
Form1::executeButton_Click 0.11 14.98 100.00 % 100.00 % 1
% with Children
Children
Routine Name Time Time with Child... % Time % with C... Hit Count
Form1::DoActionC 0.11 10.19 4.54 % 68.52 % 10
MessageBox::Show 0.00 2.41 0.03 % 16.18 % 1
Form1::Sleep 1.73 1.73 72.36 % 11.60 % 160
Form1::DoActionA 0.50 0.50 20.96 % 3.36 % 1
Monitor Call Tree Details Event View Call Graph PE Reader Disassembler

Report panel in AQtime integrated into Borland Developer Studio

Organization

A summary of profiling results is shown in the Summary panel. It helps you quickly find routines with poor performance and determine which code snippets of your application should be optimized.

Most profilers organize results into categories. For instance, the results of the Allocation profiler are organized into the *Classes* and *Objects* categories. Within categories, AQtime stores profiling results for each thread (AQtime also stores results for the entire application). You can select the desired category and thread in the Explorer panel or from the **Result Items** toolbar item (by default, this item is hidden):



The Report panel shows results for the selected thread and category. The other panels display additional profiling results. The Report panel shows a table where each row corresponds to a routine, line, class, module or object that has been profiled. When you select a row, other panels are updated to display information for that routine or line (not all the panels apply to all profilers):

- The Call Graph panel displays the call information for the selected routine.
- The Call Tree panel displays all call “paths”for the routine.
- The Details panel holds additional profiling results for the routine, e.g. the call stack, “child”and “parent”routines, etc.
- The Editor panel displays the source code of the selected routine.
- The Disassembler panel displays the binary code of the selected routine.
- The Event View panel is unaffected; it simply logs events that occurred during the profile run.

Note that some AQtime panels hold footers that show summary values for data stored in panel columns. By default, the footer of a column displays the summary value that is calculated against all panel rows. However, if you select two or more rows in the panel, AQtime will recalculate the summary and the footer will display summary values for selected rows only.

Managing results

- You can sort results by one or several columns.
- You can group results by one or several columns.
- You can search results using the Find dialog or the Incremental Search feature.
- You can add summary fields.
- You can filter results.
- Better yet, you can apply a result view from the many pre-defined ones, or from those you define yourself. A *result view* combines a filter, layout and settings of columns in AQtime panel and in the Editor’s grid and layout of panes in the Details panel.
- Using the Explorer panel, you can compare current results with previous ones to find the effects of changes you made in the application (or in the way you ran it). In addition, you can merge several results to collect mass statistics. The Explorer panel lets you organize profiling results like files and folders in Windows Explorer.

Transferring results

Profiling results can be:

- copied to the Clipboard,
- printed using the Print Preview Form,
- exported to text, Excel, html or xml formats (see Exporting Results).

More usability features

- While you use the Report or the Details panels, AQtime records your movements from item to item. You can come back to something you selected previously, and then return to where you jumped back from, as with an Internet browser. Use the **Display Previous** and **Display Next** buttons on the Report toolbar to do this. See *AQtime Panels* for more information.
- AQtime's visual means for arranging grids apply to the display of results, of course. You can:
  - change column width and ordering,
  - hide or show columns (not all columns are displayed by default).

© 2010 AutomatedQA Corp.

AQtime Panels

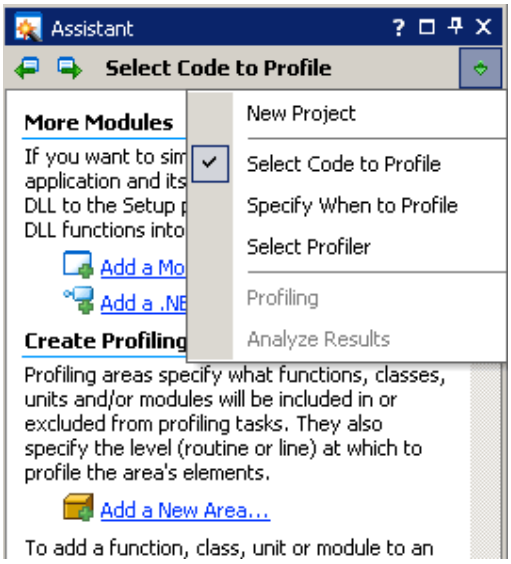
Assistant Panel

The Assistant panel helps you get started with AQtime quicker. The panel includes several pages that are brought up depending on the current context. These pages display information to assist you in working with AQtime. For instance, when you are reviewing profiler results, the Assistant panel displays the Analyze Results page with information and links to help topics that hold information that concern viewing results.

By default, the panel is displayed to the right of AQtime's, Visual Studio's or Borland Developer Studio's main window. If the Assistant panel is not visible, you can bring it up by doing any of the following:

- *AQtime standalone:*
  - Select **Assistant** from the **View | Other Panels** or **Help** menu.
  - Select **Assistant** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.
- *AQtime integrated into Visual Studio:*
  - Select **Assistant** from the Select Panel dialog, which is called by selecting **Profile | Panel List**.
  - Select **Assitant** from Visual Studio's **Solution Explorer**.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Assistant** from the **View | Profile Windows | Other** menu.

To switch between pages in the Assistant panel, press the and buttons at the top of the panel or select the desired page name from the menu:


Assistant
Select Code to Profile
More Modules
If you want to sim
application and its
DLL to the Setup
DLL functions into
Add a Mo
Add a .NE
Create Profiling
Profiling areas specify what functions, classes, units and/or modules will be included in or excluded from profiling tasks. They also specify the level (routine or line) at which to profile the area's elements.
Add a New Area...
To add a function, class, unit or module to an
New Project
Select Code to Profile
Specify When to Profile
Select Profiler
Profiling
Analyze Results

Currently, the Assistant panel includes the only customizable option, *Context-sensitive*. It specifies whether AQtime automatically changes the Assistant panel page according to changes in the context. You can check or change this option using the **Assistant Options** dialog. To call it, do any of the following:

- *AQtime standalone:*
  - Select **Options** from the context menu of the Assistant panel.
  - Select **Options | Options** from AQtime's main menu and then choose **Services | Assistant** in the ensuing Options dialog.
- *AQtime integrated into Visual Studio:*
  - Select **Options** from the context menu of the Assistant panel.
  - Right-click **Assistant** in Visual Studio's **Solution Explorer** and select **Properties** from the context menu.
  - Select **Tools | Options** from the main menu of Visual Studio and then choose the **AQtime | Services | Assistant** in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Options** from the context menu of the Assistant panel.
  - Select **Profile | Options** from the main menu of Borland Developer Studio and then choose **Services | Assistant** in the ensuing Options dialog.

Call Graph Panel

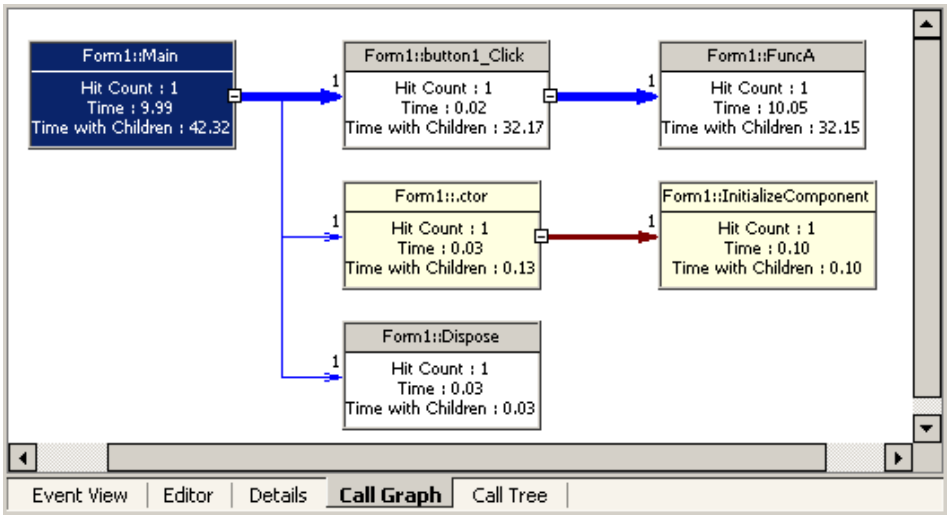
Call Graph Panel - Description

The **Call Graph** panel can be used with the Performance, Function Trace and Allocation profilers. For the **Performance** and **Function Trace** profilers, Call Graph displays the hierarchy of calls for the routine clicked in the **Report** panel. For the **Allocation** profiler, the Call Graph displays the hierarchy of object references for the object clicked in Report. To display the Call Graph panel, do any of the following:

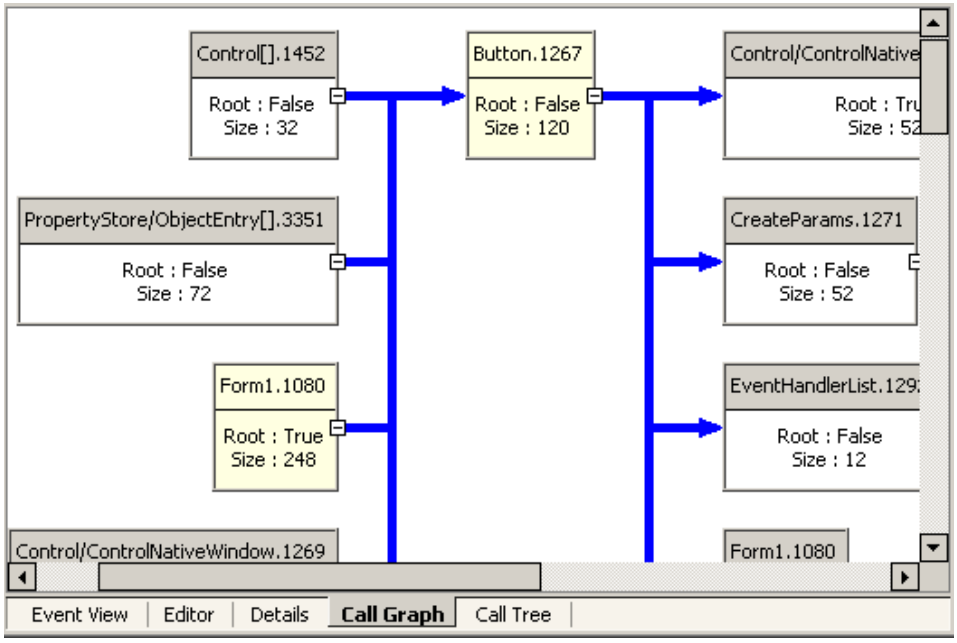
- *AQtime standalone:*
  - Select **Call Graph** from the **View | Other Panels** menu.
  - Select **Call Graph** from the Select Panel dialog, which is called by selecting **View | Select Panel**.
  - Select **Call Graph** from the **Assistant** panel.
- *AQtime integrated into Visual Studio:*
  - Select **Call Graph** from the Select Panel dialog, which is called by selecting **Profile | Panel List**.
  - Select **Call Graph** from Visual Studio's **Solution Explorer**.
  - Select **Call Graph** from the **Assistant** panel.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Call Graph** from the **View | Profile Windows | Other** menu.
  - Select **Call Graph** from the **Assistant** panel.

In the upper part of the rectangle you can see the routine name, and in the lower part, the profiling results:

© 2010 AutomatedQA Corp.



In the object reference hierarchy, each object is represented as a rectangle and the arrows show the sequence of references between objects. In the upper part of the rectangle you can see the class name and instance number of the given object, and in the lower part, the object size.



All panel settings are divided into two large groups: options specific to the current profiler and options common for all profilers. To modify options specific to the current profiler, use the **Customize Call Graph** dialog (to call it, select **Customize** from the Call Graph context menu). To modify options that do not depend on the profiler, use the **Call Graph Options** dialog (it appears upon selecting **Options** from the context menu).

The Customize Call Graph dialog lets you select:

- Result values to be displayed,
- Result format,
- Sorting order, etc.

If the *Show results as hint* option is enabled, profiling results are displays as a hint when you move the mouse over a rectangle. If this option is off, profiling results are displayed within the rectangle.

In the Call Graph Customization dialog, you can also select what values should be displayed on the graph's arrows. For instance, if you select **Hit Count** (for the Performance profiler) then the numbers at the beginning of the arrows (from parent routines), specify how many times the highlighted routine was called from the parent one. Those at the arrow heads (towards child routines) specify how many times a child was called from the highlighted routine.

The **Customize Call Graph** dialog offers one more option, *Show cycling connections*, which you can use to specify how Call Graph should show recursive function calls and object references.

The number of "child" and "parent" levels in the chart is specified by two options in the **Call Graph Options** dialog: *Number of child levels* and *Number of parent levels*. Increase these settings to show more of the routine call hierarchy. Decrease them to simplify the display. If you want to view the entire tree of routine calls or object references, use the **Call Tree** panel.

Like the Call Tree panel for the Performance profiler, the Call Graph can display the *critical path* for the call hierarchy currently displayed. The critical path, which is highlighted in bold, is the "longest" route among all of the call routes in the graph. For instance, the route in which **Time with Children** summarized for all of the routine calls that belong to this route is the longest of all routes of the graph. The critical path allows you to determine which series of routine calls has the most significant effect on the performance of the profiled application. Optimizing routines that belong to the critical path will help you acheive optimization of the entire application faster. Note that since the Call Graph panel normally displays only part of the entire call tree, the critical path displayed in the graph is local, applicable to the current graph only. To get the critical path for the entire application, turn to the Call Tree panel.

To display the critical path in the Call Graph, enable the *Show critical path for column* option in the **Customize Call Graph** dialog. You can also specify which parameter will be used to calculate the "length" of the call route. To do this, select the desired column in the same box of the Customize Call Graph dialog.

Double-clicking on a routine (or object) rectangle in the panel has the same effect as clicking on that routine (object) in the Report panel – the Call Graph panel updates to highlight the clicked routine (object) and show its parents and children, and other panels update accordingly (**Editor**, **Details**, etc. See *AQtime Panels*).

The context menu has **Go to Child**, **Go to Parent** and **Go to Current Item** to help you navigate the hierarchy when it is large. It also has **Zoom In**, **Zoom Out**, **Undo Zoom** and **Fit** to help you get the view you need.

You can quickly navigate to the graph of routine calls (object references) which is built for the selected routine (object). To do this, simply double-click the rectangle of the desired routine (object) in the graph.

If the graph displays a hierarchy of routine calls, you can also navigate to the routine declaration in the source file. To do this, right-click within the routine's rectangle and select **Show Source** in the ensuing context menu. AQtime will switch to the Editor and open the source file where the corresponding routine is declared.

To export the graph to a metafile, select **Export to Metafile** from the context menu.

To print the Call Graph contents, select **Print** or **Print Preview** from the context menu. **Print** will open the printer selection dialog and send the Call Graph to printer. **Print Preview** will open the Print Preview Form where you can customize the report to be printed.

Call Graph Panel Options

The **Call Graph** panel offers several customizable options. They are divided into two groups: those that depend on the current profiler and those that do not depend on the current profiler.

To check or modify profiler-dependent options, use the **Customize Call Graph** dialog. The rest of this topic explains profiler-independent options.

You can check or change these options using the **Call Graph Options** dialog. To call it, do any of the following:

- *AQtime standalone:*
  - Select **Options** from the context menu of the Call Graph panel.
  - Select **Options | Options** from AQtime's main menu and then choose **Analysis Results | Call Graph** on the left of the ensuing Options dialog.
- *AQtime integrated into Visual Studio:*
  - Select **Options** from the context menu of the Call Graph panel.
  - Right-click **Call Graph** in Visual Studio's **Solution Explorer** and select **Properties** from the context menu.
  - Select **Tools | Options** from the main menu of Visual Studio and then choose **AQtime | Analysis Results | Call Graph** on the left of the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Options** from the context menu of the Call Graph panel.
  - Select **Profile | Options** from the main menu of Borland Developer Studio and then choose **Analysis Results | Call Graph** on the left of the ensuing Options dialog.

The following options are available:

- **Highlight** - When the mouse points to a link between rectangles, this option specifies whether the linked rectangles are highlighted to make it clearer that they are the ones linked.
  - **Active** - Sets if any highlighting is done.
  - **Child item color** – Sets the highlight color for the child-function (child-object) rectangle.
  - **Parent item color** – Sets the highlight color for the parent-function (parent-object) rectangle.
- **Item rectangle background**
  - **Header part** – Sets background color for the upper part of function (object) rectangles.
  - **Details part** – Sets background color for lower part of function (object) rectangles.
- **Show pointing-hand cursor** – Sets whether the mouse cursor will change to a pointing hand over areas (lines and rectangles) that can be clicked on to switch to new details.
- **Show results as hint** – If this option is enabled, profiling results are displayed as a hint. Otherwise, they are displayed in the lower portion of the routine (object) rectangle.
- **Number of parent levels** – Sets the depth of parent calls (parent object references) that will be displayed for each function (object).
- **Number of child levels** – Sets the depth of child calls (child object references) that will be displayed for each function (object).

Call Tree Panel

Call Tree Panel - Description

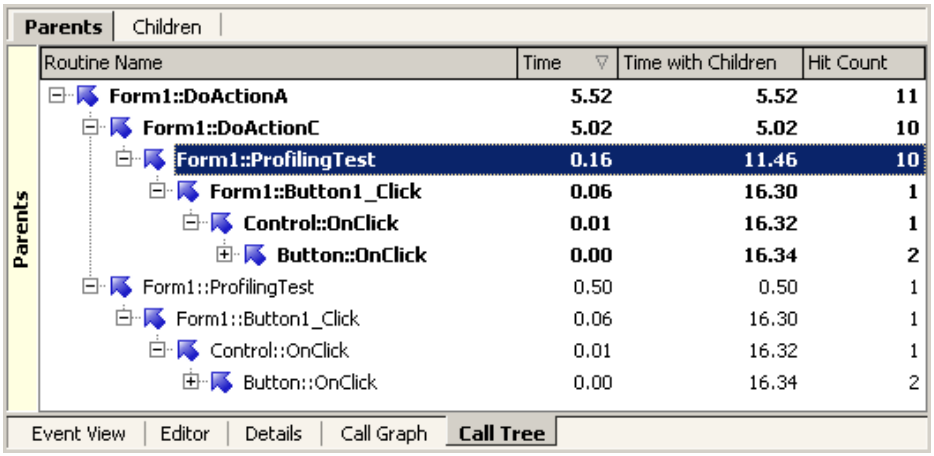
When the **Report** panel displays the profiling results for the **Performance** and **Function Trace** profilers, the **Call Tree** panel displays the hierarchy of calls for the routine clicked in Report. When Report displays profiling results for the **Allocation** profiler, the Call Tree displays the hierarchy of references between objects for the object clicked in Report. Unlike the **Call Graph**, the Call Tree panel represents information as a tree view. To display the panel, do any of the following:

- *AQtime standalone:*
  - Select **Call Tree** from the **View | Other Panels** menu.
  - Select **Call Tree** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.
  - Select **Call Tree** from the **Assistant** panel.
- *AQtime integrated into Visual Studio:*
  - Select **Call Tree** from the **Select Panel** dialog, which is called by selecting **Profile | Panel List**.
  - Select **Call Tree** from Visual Studio's **Solution Explorer**.
  - Select **Call Tree** from the **Assistant** panel.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Call Tree** from the **View | Profile Windows | Other** menu.
  - Select **Call Tree** from the **Assistant** panel.

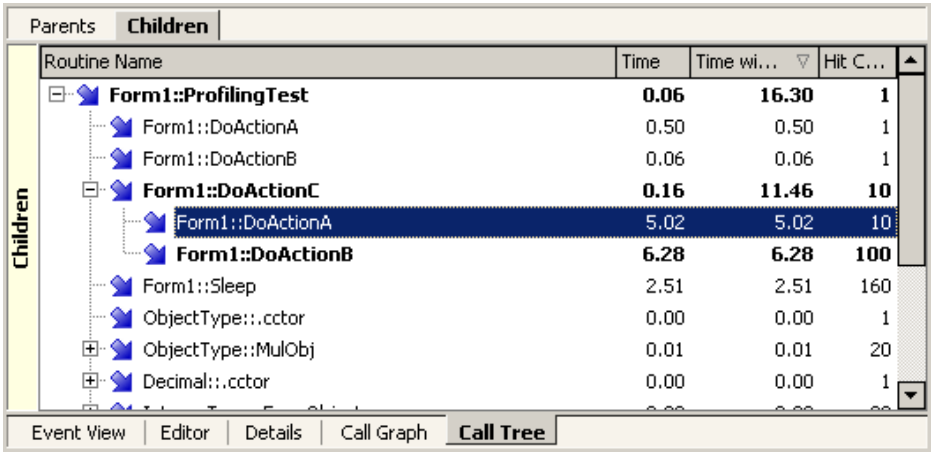
The appearance of the Call Tree panel depends on the selected profiler.

Performance Profiler

For this profiler, the Call Tree panel consists of two panes: *Parents* and *Children*. The **Parents** pane displays the tree of calls that led to a call to the given routine.



Similarly, the **Children** pane displays the tree of calls started by the given routine.



For the routine chosen in the Report panel, both panes display all the routes of routine calls that were used during profiling. So you can easily retrace the desired route.

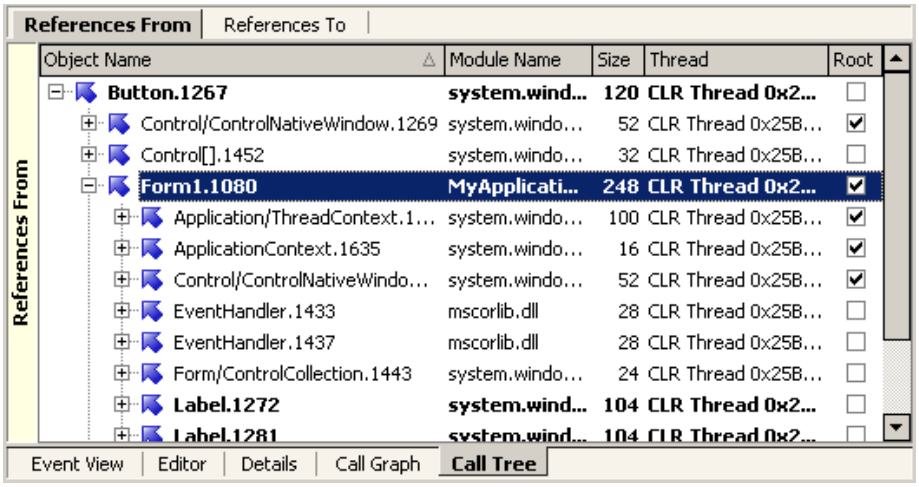
The panes of the **Call Tree** can display information in the same columns as those used in the **Calls** pane of the **Details** panel for the **Performance** profiler.

Note that in both Parents and Children panes the routine selected in the Report panel is the start node of the panes' trees. As a result, these panes display their call trees in different chronological order. In the Children pane, the call hierarchy is displayed in the direct order (a routine that was called by another routine is displayed as a child of that routine's node in the tree), while in the Parents pane, the call hierarchy is displayed in the reverse order (a routine that called another routine is displayed as a child of that routine's node in the tree).

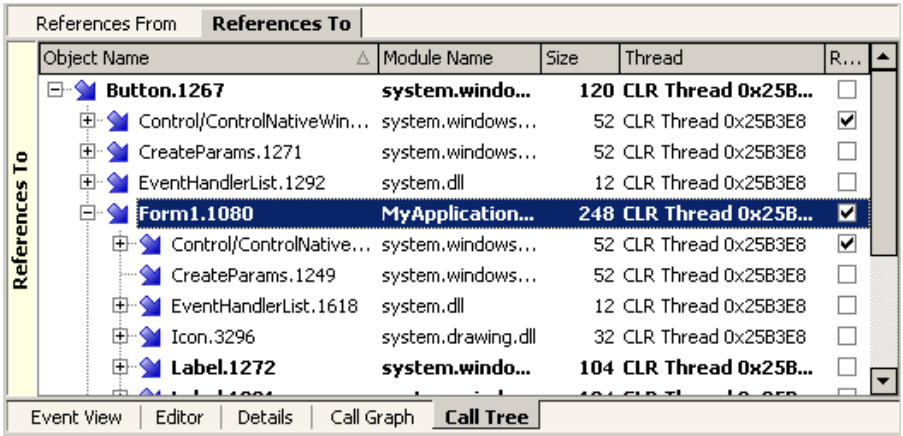
Allocation Profiler

For this profiler, the **Call Tree** panel can display results when profiling results are displayed in the **Report** panel. The Call Tree panel consists of two panes: *References From* and *References To*.

The **References From** pane displays the following tree of references between objects: what objects held references to the given object, what objects referred to these objects and so on.



The **References To** pane displays a different tree of references between objects: what objects the given object had references to, what objects these objects referred to and so on.



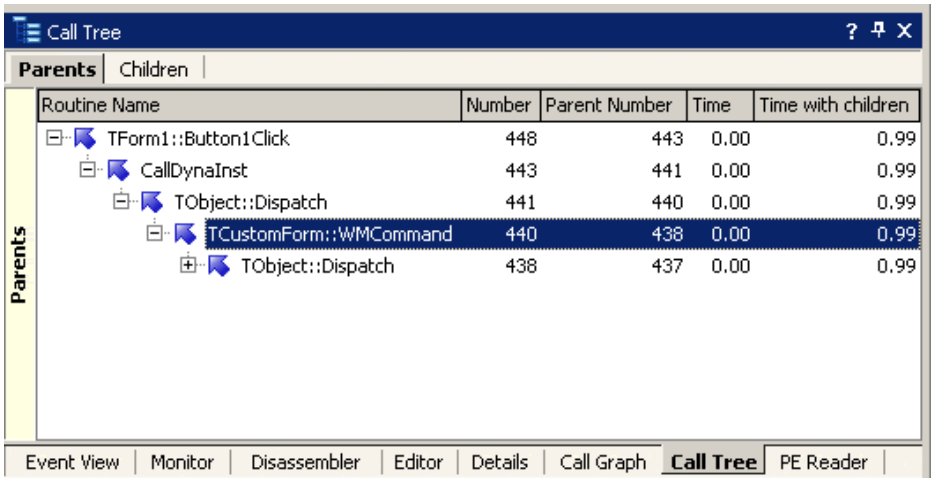
For the object chosen in Report, the panes display all of the object references that were created during profiling.

The panes of the Call Tree can display information in the same columns as those used in the **References To** and **References From** panes of the **Details** panel for the **Allocation** profiler.

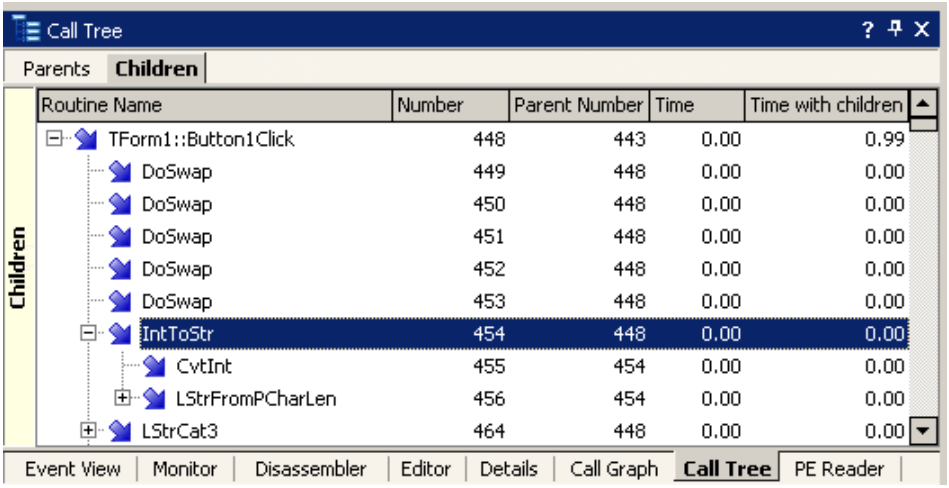
Note that in both panes the object selected in the **Report** panel is the start node in the panes' tree. As a result, these panes display their object reference trees in different chronological order. In the References To pane, object references are displayed in the direct order (an object that is referred to by another object is displayed as a child of that object's node in the tree), while in the References From pane, object references are displayed in the reverse order (an object that refers to another object is displayed as a child of that object's node in the tree).

Function Trace Profiler

When the **Call Trace** category is selected for the Function Tracer results you can review the routine call hierarchy in the CallTree panel. For this profiler, the Call Tree panel consists of two panes: *Parents* and *Children*. The **Parents** pane displays the tree of calls that led to the given routine being called.



Similarly, the **Children** pane displays the tree of calls started by the given routine.



For the routine chosen in the Report panel, both panes display all the routes of routine calls that were used during profiling. So you can easily retrace the desired route.

The panes of the Call Tree display information in the following columns that do not depend on the active counter:

Columns (in alphabetical order)	Description
Number	The number of routine calls.
Parent Number	The number of the routine that called the given routine.

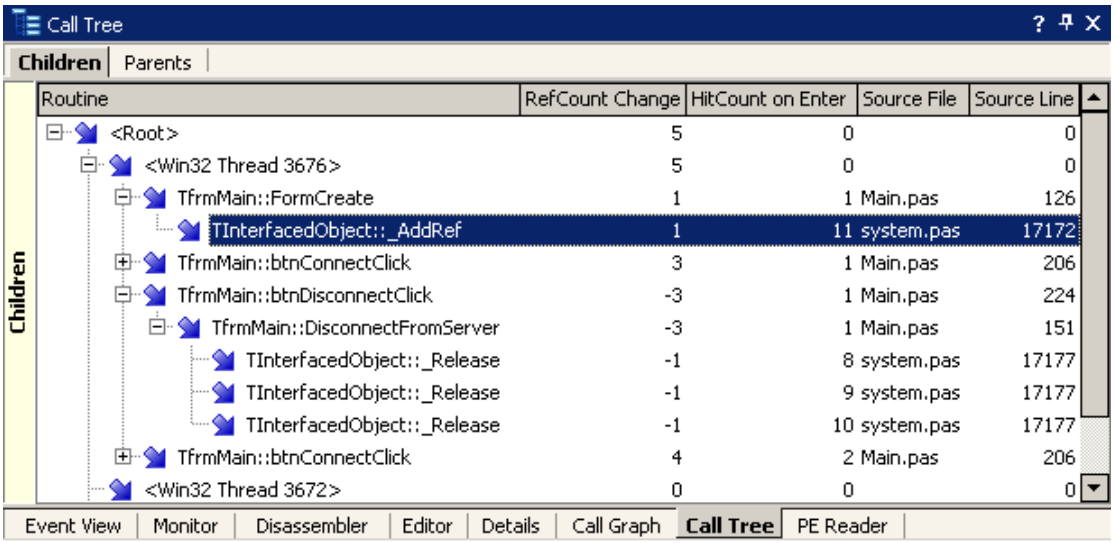
Routine Name	Name of the routine.

Additionally several counter-specific columns are displayed. These columns contain the same values as counter-specific columns of the Report panel. See the column descriptions in the *Function Trace - Report Panel Call Trace Category* topic.

Note that in both Parents and Children panes the routine selected in the Report panel is the start node of the panes' trees. As a result, these panes display their call trees in different chronological order. In the Children pane, the call hierarchy is displayed in the direct order (a routine that was called by another routine is displayed as a child of that routine's node in the tree), while in the Parents pane, the call hierarchy is displayed in the reverse order (a routine that called another routine is displayed as a child of that routine's node in the tree).

Reference Count Profiler

When the **Objects** category is selected for Reference Count results, you can use the Call Tree panel to trace how object references were created or released. The panel consists of two panes: **Parents** and **Children**, but actualy only the latter pane is used. The **Children** pane displays the tree of calls that led to creation or deletion of references to the object chosen in the Report panel.



The panes of the Call Tree panel display information in the following columns:

Columns (in alphabetical order)	Description
Hit Count on Enter	The routines' call number that signifies when it was placed in the stack.
RefCount Change	The number of times the reference counter was modified.
Routine	The name of the routine.
Source File	The name of the source file for the routine. The values for this column are read from the application's debug info. If the debug info does not contain information on the file name, the column is empty.
Source Line	The source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.

Like the **Call Graph** panel, each pane of the **Call Tree** can display the *critical path* for the given tree of calls or object references. The critical path (in bold) is the "longest" route in the tree. For instance, for the Performance profiler, the critical path can be the route in which the sum of the **Time with Children** values is the largest in the tree. Another example is for the Allocation profiler, for which the critical path can be the route in which the sum of the **Size** values is the largest in the tree.

The critical path allows you to determine which series of routine calls has the most significant effect on the performance of the profiled application. Optimizing routines that belong to the critical path will help you reach optimization of the entire application faster.

You can specify which parameter will be used to calculate the "length" of the route. To do this, select **Customize** from the context menu and in the resulting **Call Tree Options** dialog select the desired column. In addition, using this dialog, you can specify if AQtime should expand nodes of the critical path and if yes how many of them should be expanded (starting from the root node).

You can manually expand or collapse all the nodes of the tree by choosing **Expand All** or **Collapse All** from the panel's context menu. Besides, you can store the panel contents to a file (an HTML, XML or text file) by selecting the **Save All** context menu item.

No matter which profiler you use the Call Tree panel for, you can easily customize which columns you want to display in the selected pane and which ones you want to hide. See *Adding and Removing Columns*. In addition, in the selected pane you can sort records by a column. This applies the specified sorting order to all of the records that belong to the same level in the tree. In addition, like in other AQtime panels that hold table information, you can specify the desired display format for a particular column. See *Changing Column Format*.

You can quickly navigate to the tree of routine calls (object references) which is built for the selected routine (object). To do this, simply double-click the desired routine or object in the tree.

Call Tree Panel Options

The Call Tree panel offers several customizable options. To check or modify them, use the **Call Tree Options** dialog. To call the dialog, do any of the following:

- *AQtime standalone*:
  - Right-click somewhere within the Call Tree panel and choose **Customize** from the context menu.
  - Select **Options | Options** from AQtime's main menu and then choose **Analysis Results | Call Tree** on the left of the ensuing Options dialog.
- *AQtime integrated into Microsoft Visual Studio:*
  - Select **Options** from the context menu of the Call Tree panel.
  - Right-click **Call Tree** in Visual Studio's Solution Explorer and select **Properties** from the context menu.
  - Select **Tools | Options** from the main menu of Visual Studio and then choose **AQtime | Analysis Results | Call Tree** on the left of the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Options** from the context menu of the Call Tree panel.
  - Select **Profile | Options** from the main menu of Borland Developer Studio and then choose **Analysis Results | Call Tree** on the left of the ensuing Options dialog.

The following options are available:

- **Hide recursion** - If this option is enabled, AQtime will not display recursive calls in the Call Tree panel which shortens the displayed paths. If this option is disabled, the recursive calls are shown.
- **Maximum call depth** - Specifies the maximum number of calls that can be shown in the panel.

Three other options deal with critical paths. The *critical path* is the longest route in the Call Tree calculated by a particular criterion.

- If the **Show critical path for column** option is enabled, the Call Tree panel highlights the critical path (with bold) for the routines or objects it displays.
- **Auto-expand critical path to depth** - If this option is enabled, AQtime expands nodes of the critical path automatically to the depth specified by the *Auto-expand depth* option.
- **Auto-expand depth** - Specifies which depth AQtime will automatically expand for the critical path. This option is only meaningful when the *Auto-expand critical path to depth* setting is enabled.

Details Panel

Details Panel - Description

The **Details** panel displays additional results for the line selected in the **Report** panel with a number of profilers. The details displayed depend on the profiler. See:

- *Allocation Profiler*
- *BDE SQL Profiler*
- *Coverage Profiler*
- *Function Trace Profiler*
- *Light Coverage Profiler*
- *Load Library Tracer*
- *Performance Profiler*
- *Reference Count Profiler*
- *Resource Profiler*
- *Static Analysis Profiler*
- *Unused VCL Units Profiler*

To display the Details panel, do any of the following:

- *AQtime standalone:*
  - Select **Details** from the **View | Other Panels** menu.
  - Select **Details** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.
  - Select **Details** from the **Assistant** panel.
- *AQtime integrated into Visual Studio:*
  - Select **Details** from the **Select Panel** dialog, which is called by selecting **Profile | Panel List**.
  - Select **Details** from Visual Studio's **Solution Explorer**.
  - Select **Details** from the **Assistant** panel.

- *AQtime integrated into Borland Developer Studio:*
  - Select **Details** from the **View | Profile Windows | Other** menu.
  - Select **Details** from the **Assistant** panel.

You can arrange the Details panel the same way you can organize other AQtime panels.

Details Panel Options

Currently, the **Details** panel offers the only customizable option:

- **Show footer** – Sets whether a footer row will be added to show totals for some columns. The columns depend on the current profiler.

You can check or change this option using the **Details Options** dialog. To call it, do any of the following:

- *AQtime standalone:*
  - Select **Options** from the context menu of the Details panel.
  - Select **Options | Options** from AQtime's main menu (this will call the Options dialog) and then choose **Analysis Results | Details** from the tree view on the left of the dialog.
- *AQtime integrated into Visual Studio:*
  - Select **Options** from the context menu of the Details panel.
  - Right-click **Details** in Visual Studio's **Solution Explorer** and select **Properties** from the context menu.
  - Select **Tools | Options** from the main menu of Visual Studio and choose the **AQtime | Analysis Results | Details** group in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Options** from the context menu of the Details panel.
  - Select **Profile | Options** from the main menu of Borland Developer Studio (this will call the Options dialog) and then choose **Analysis Results | Details** from the tree view on the left of the dialog.

Disassembler Panel

Disassembler Panel - Description

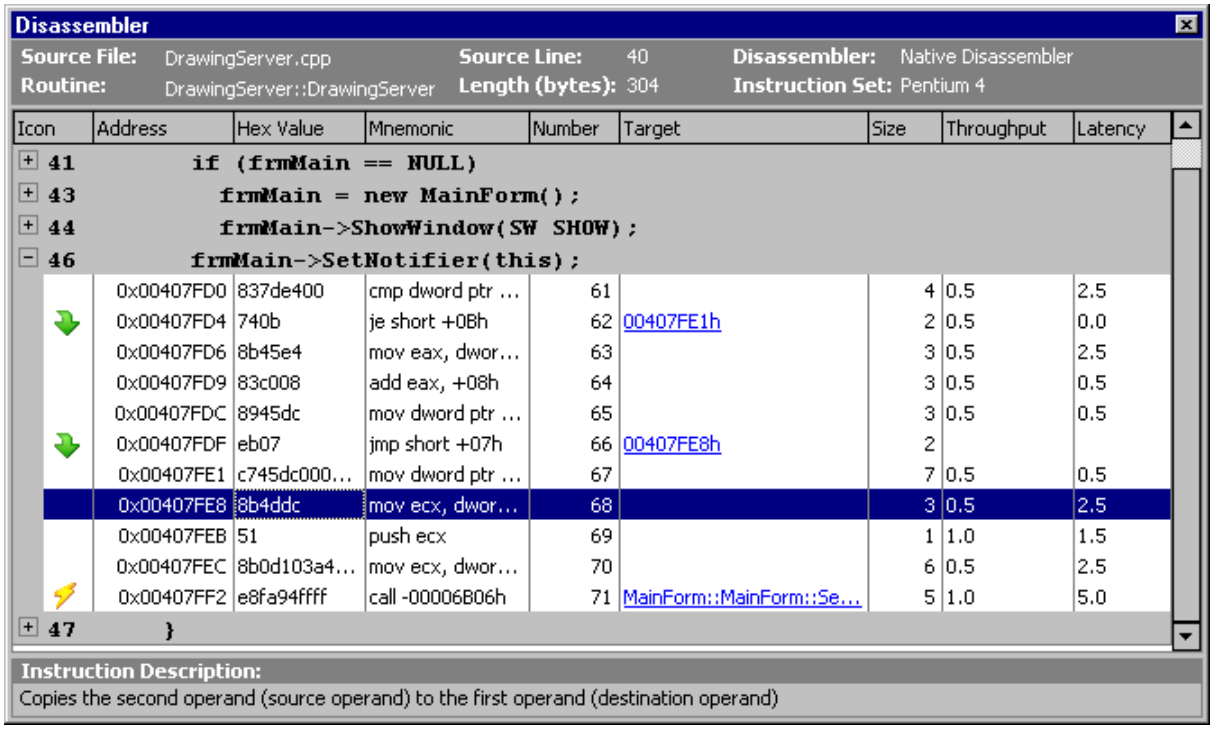
The purpose of the **Disassembler** panel is to allow you to check the binary code of your routines independent of the compiler, version or library behind this code. The Disassembler panel is updated when you double-click (click) on a routine in one of the AQtime panels: Report, Details, Event View, Setup, Call Graph, Call Tree, PE Reader or Summary. The panel can display the disassembly for both managed and unmanaged code.

To display the Disassembler panel, do any of the following:

- *AQtime standalone:*
  - Select **Disassembler** from the **View | Other Panels** menu.
  - Select **Disassembler** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.
  - Select **Disassembler** from the **Assistant** panel.

- *AQtime integrated into Visual Studio:*
  - Select **Disassembler** from the **Select Panel** dialog, which is called by selecting **Profile | Panel List**.
  - Select **Disassembler** from Visual Studio's **Solution Explorer**.
  - Select **Disassembler** from the **Assistant** panel.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Disassembler** from the **View | Profile Windows | Other** menu.
  - Select **Disassembler** from the **Assistant** panel.

Here is a sample view of the Disassembler panel --


Disassembler
Source File: DrawingServer.cpp
Source Line: 40
Disassembler: Native Disassembler
Routine: DrawingServer::DrawingServer
Length (bytes): 304
Instruction Set: Pentium 4
Icon | Address | Hex Value | Mnemonic | Number | Target | Size | Throughput | Latency
41 if (frmMain == NULL)
43 frmMain = new MainForm();
44 frmMain->ShowWindow(SW_SHOW);
46 frmMain->SetNotifier(this);
0x00407FD0 | 837de400 | cmp dword ptr ... | 61 | | 4 | 0.5 | 2.5
0x00407FD4 | 740b | je short +0Bh | 62 | 00407FE1h | 2 | 0.5 | 0.0
0x00407FD6 | 8b45e4 | mov eax, dwor... | 63 | | 3 | 0.5 | 2.5
0x00407FD9 | 83c008 | add eax, +08h | 64 | | 3 | 0.5 | 0.5
0x00407FDC | 8945dc | mov dword ptr ... | 65 | | 3 | 0.5 | 0.5
0x00407FDF | eb07 | jmp short +07h | 66 | 00407FE8h | 2 | | |
0x00407FE1 | c745dc000... | mov dword ptr ... | 67 | | 7 | 0.5 | 0.5
0x00407FE8 | 8b4ddc | mov ecx, dwor... | 68 | | 3 | 0.5 | 2.5
0x00407FEB | 51 | push ecx | 69 | | 1 | 1.0 | 1.5
0x00407FEC | 8b0d103a4... | mov ecx, dwor... | 70 | | 6 | 0.5 | 2.5
0x00407FF2 | e8fa94ffff | call -00006B06h | 71 | MainForm::MainForm::Se... | 5 | 1.0 | 5.0
47 }
Instruction Description:
Copies the second operand (source operand) to the first operand (destination operand)

You can arrange the Disassembler panel the same way you organize other AQtime panels.

When *Show panel header* panel option is enabled, the panel displays a header that lists the names of the source file and routine, as well as the instruction set, disassembly type (native or managed), routine length (in bytes) and the number of the source file line at which the routine code starts.

The *Display code* panel option specifies whether the panel displays the source code along with the assembler instructions. The option can have one of the following values --

Value	Description
Assembler only	The panel displays assembler instructions, one per line.
Assembler and source	The panel displays the source code and each source line becomes a node that can be expanded into its assembler instructions. Note that in this case, the Disassembler panel does not display lines, which do not have assembler instructions. That is it does not display empty lines, comments, etc.
Assembler and full source	Similar to Assembler and source. The only difference is that the source code includes the lines for which there is no debug

	information.

The **Instruction Description** box at the bottom of the panel displays the description of the selected assembler instruction, when the *Show instruction description* option is on.

Note that binary code for a different processor can mean different commands. That is, the same sequence of bytes can be different instructions for different processors, for example, for Pentium IV and AMD K6. Use the *Processor family* option to specify what type of CPU the executable was compiled for.

The panel columns are as follows --

Column (in alphabetical order)	**Description**
Address	The starting address of the instruction in memory. The format of the address depends upon the *Show addresses as RVA* option.
ANSI Value	Alternative translation of the binary code into ANSI-string format instead of disassembly. Quickly marks out string constants embedded in the code.
Hex Value	The hexadecimal string of bytes composing the instruction.
Icon	Icon identifying certain instructions, such as jump, return, call, dd, dw and db.
Latency	The latency characteristic of the assembler instruction (in processor cycles). If the asterisk symbol (*) is displayed next to the given value, this means that the value was not accurately calculated.
Line No	The source line number in the source file.
Mnemonic	Assembler instruction matching the Hex string.
µOps	The count of micro-instructions (micro-operations) in the assembler instruction. Note that for Pentium IV this value cannot be calculated. Therefore, this column displays empty values and its summary displays the astersik symbol (*).
Number	The number of the assembler instruction from the beginning of the routine.
Offset	Instruction's position relative to the beginning of the routine's code in memory.
Size	The instruction's size in bytes (i.e., length of its hex string).
Target	Next address to be executed after a jump, a call, etc. The target address is displayed as a hyperlink which you can click to navigate to the desired address quickly. If this address lies within the routine's address space it is displayed as a number. Otherwise, the name of the routine at the target address along with the name of its class are given.
Target Module	Name of the module (EXE, DLL, OCX, etc.) that contains the target routine.
Throughput	The throughput characteristic of the assembler instruction (in processor cycles). If the asterisk symbol (*) is displayed next to the given value, this means that the value was not accurately calculated.

You can copy the selected instructions to the clipboard by clicking **Copy** from the context menu. You can also save the selected instructions to a text file by using **Save Selection**. The usual Ctrl and Shift commands

allow for multiple selections (see *Selecting Several Records in a Panel*). To save all instructions displayed in the panel, select **Save All**.

By double-clicking on the instruction records that hold the or jump, return or call icons, you can switch to the disassembly for the target routine. (The icon indicates instructions identified as data.) When using these direct jumps, the Disassembler panel tracks your movements among routines. The **Display Previous** and **Display Next** context menu items let you move back and forth among the routines you have previously viewed.

Disassembler Panel Options

The **Disassembler** panel offers several customizable options. You can check or change these options using the **Disassembler Options** dialog. To call it, do any of the following:

- *AQtime standalone:*
  - Select **Options** from the context menu of the Disassembler panel.
  - Select **Options | Options** from AQtime's main menu and choose the **Services | Disassembler** group in the ensuing Options dialog.
- *AQtime integrated into Visual Studio:*
  - Select **Options** from the context menu of the Disassembler panel.
  - Right-click **Disassembler** in Visual Studio's **Solution Explorer** and select **Properties** from the context menu.
  - Select **Tools | Options** from the main menu of Visual Studio and choose the **AQtime | Services | Disassembler** group in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Options** from the context menu of the Disassembler panel.
  - Select **Profile | Options** from the main menu of Borland Developer Studio and choose the **Services | Disassembler** group in the ensuing Options dialog.

All options are divived into two large groups, Disassembler Panel and Disassemblers, which you select on the left of the dialog.

- **Disassembler Panel** - Options of this group specify the appearance and behaviour of the Disassembler panel, which is the disassembly display in AQtime.
  - **Grid settings**
    - **Auto expand** - This option only applies when the *Display code* option is set to *Assembler and source* or *Assembler and full source*. If *Auto expand* is on, the Disassembler panel automatically expands the nodes, which correspond to source lines. Otherwise, these nodes remain collapsed.
    - **Show summary** - Sets whether a footer row will be added to show totals for the Size, Latency, Ops and Throughput columns.
    - **Show source line summary** - Sets whether an extra line will be added to the disassembly for each source line (when displayed), in order to show Size, Latency, Ops and Throughput totals for the source line.
    - **Grid lines display mode** - Specifies the mode of displaying grid lines between columns and rows. Available values: *Both* (both vertical and horizontal grid lines are shown), *None* (no grid lines are shown), *Vertical* (only vertical grid lines are shown) and *Horizontal* (only horizontal grid lines are shown).

- **Content settings**
  - **Disassembly lines** - Options that affect the appearance of the instruction lines in the panel's grid.
    - **Word wrap** - If this option is enabled, the Disassembler panel wraps text in its cells. Otherwise, the panel displays ellipses to show that some part of the text is hidden.
    - **Upper case mnemonics** - Sets whether assembler instructions will be shown in upper case, rather that lower.
    - **Background color** - Sets the background color for the disassembly lines.
  - **Source lines** - Options that affect the appearance of the source code lines in the panel's grid.
    - **Tab width** - Sets the width in spaces of a tab character. 2 is default. 0 means that the standard Windows tab width will be used.
    - **Source line color** - Sets the background color for the source code lines.
- **Show panel header** - Sets whether to display special information about the selected routine at the top of the panel.
- **Show instruction description** - Sets whether to display description of the selected instruction at the bottom of the panel.
- **Display code** - Specifies how AQtime displays source code in the panel. This option can have one of the following values:
  - *Assembler only* - The panel displays only the assembler instructions, one per line.
  - *Assembler and source* - The panel displays the source code. Each source line appears as a node, which can be expanded to display its assembler instructions. Note that Disassembler displays only the source lines for binary code that exists. That is, the panel does not display empty lines, comments, etc.
  - *Assembler and full source* – Similar to *Assembler and source*. However, if this value is set, the Disassembler panel displays all of the source lines, including those for which the compiler did not generate assembler instructions: Empty lines, comments, etc.

- **Disassemblers** - Options of this group are used to configure disassemblers that are installed in AQtime.
  - **x86 Disassembler** - Options that affect the x86 native code disassembler.
    - **Processor family** - The same sequence of hex codes may correspond to different instructions on different processors. The Processor family option specifies the processor type for which the executable was compiled. Use this option to provide proper disassembly info, because if the disassembler confuses one instruction, it will also confuse other instructions that follow the one that causes the confusion.
    - **Show addresses as RVA** - This option specifies what format should be used to display addresses in the **Address** and **Target** columns of the Disassembler panel. The address of each routine consists of two components: the base address of the module, which is the address where the module is loaded in memory, and the offset of the routine relative to this base address. The offset is also called a relative virtual address (RVA). If *Show addresses as RVA* is enabled, the Address and Target columns display only relative virtual addresses. Otherwise, they display the full routine addresses, i.e. the base address + the offset. Note that the base address can be determined only after the module has been loaded into memory. The preferred loading address is used as the base one. The preferred loading

address is specified in the header of the executable. To find it in AQtime, check the **Optional header** section on the **PE Information** page of the **PE Reader** panel.

Editor Panel

Editor Panel - Description

The **Editor** panel displays source code in AQtime running as a standalone application. To display the Editor panel, do any of the following:

- Select **Editor** from AQtime's **View | Other Panels** menu.
- Select **Editor** from the Select Panel dialog, which is called by choosing **View | Select Panel** from AQtime's main menu.
- Select **Editor** from the Assistant panel.

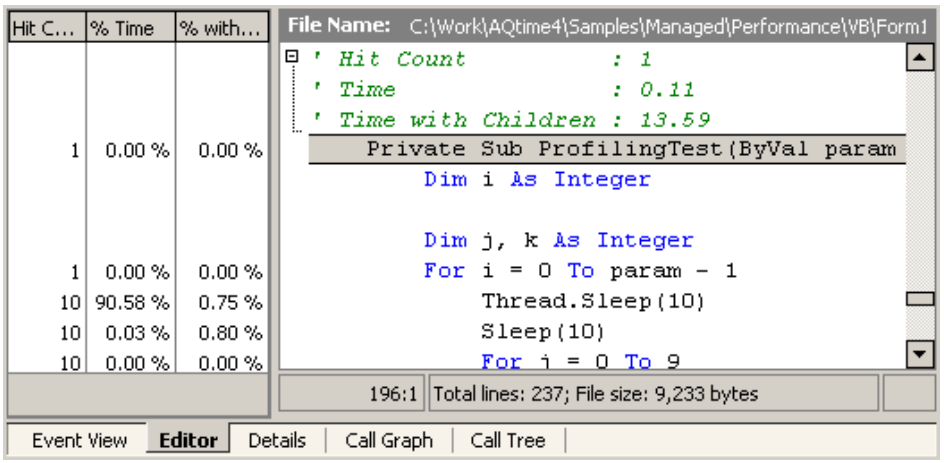
The Editor panel updates to display the source for the last routine double-clicked in the Setup, Summary, Report, Details, Call Graph, Call Tree, PE Reader or Event View panel.

To view profiling results of a routine whose source code is currently displayed in the Editor, in the Report panel, right-click somewhere in the routine's code and select **Synchronize Report** from the context menu.

The Editor displays source code unless the routine's source file cannot be found on the search path. The application sources may not be displayed in the following cases:

- Your executable was not compiled with debug information: both managed and unmanaged modules must be compiled with debug information in order for AQtime to be able to find information on the routine's source file. See *How AQtime Profilers Use Metadata and Debug Information*. To solve the problem, please compile your application with debug information.
- The debug info file is absent. This may happen, for instance, if you compiled your application with debug information, but added another file (it has the same name, but does not include debug information) to your AQtime project. Please make certain that you included the appropriate module.
- The source file is not displayed when you select a class in the Report, Details or another AQtime panel. This happens because debug information does not hold information about source files for classes.
- The source file has been removed or it has never been created (some compilers, for instance, Visual Basic .NET, can compile the executable without saving its sources to a disk file).
- The source file cannot be found in the search path. To solve the problem, you can use two links - **Project Search Directories** and **Search Directories** (AQtime shows them when it cannot find source files). A click on these links opens the **Project Search Directories** and **Search Directories** dialogs where you can specify the path to the desired file. You can also open any source file simply by dragging it to the panel from Windows Explorer, or by using **Open** on the panel's context menu.

Here is a sample view of the Editor panel:



The Editor panel uses **syntax color highlighting** - different fonts and colors for different code elements. This makes it easier to distinguish and locate these elements. For highlighting, AQtime uses syntax highlighting settings of the development tool associated with the file being displayed. Note that syntax highlighting settings set both font appearance and font language. For more information, see the description of the *Highlighting settings*.

The Editor's grid (it is on the left side of the panel) displays profiling results for routines and source code lines. The grid is shown if a source file holds one or more routines that were profiled at line level. Like in other panels that contain grids, you can move the columns within this grid as well as add needed columns to the grid or hide them (see *Arranging Columns, Lines and Panels*). You can also configure the format of grid columns. To do this, select **Format Columns** from the context menu and use the ensuing Format Columns dialog.

The grid's footer displays a summary value for selected rows. If there are no rows selected, the footer shows summary for all rows.

The Editor can also display profiling results for routines in source code. The results are displayed as comments before the routine's source code. You can collapse and expand the block of comments by clicking the **+** and **-** buttons in the Editor's gutter. To collapse and expand all blocks, use the **Edit | Collapse All** and **Edit | Expand All** menu items.

To customize what results to display in these comments, use the Customize Comments dialog, which is called by selecting **Customize Comments** from the Editor's context menu.

To save the results that are shown as comments to the source file, right-click within the Editor and select **Save Results to Source File** from the context menu. AQtime will append the appropriate comments before each profiled rotuine in the source file.

Note: After saving results to the source file, the Editor will display two sets of profiling results for the routine: one of these sets will be the comments generated by the Editor and another one - the results saved to the source file.

You may need to refresh the contents of the Editor panel when the file that is currently open in the panel is changed by another program. To do this, select **Refresh** from the context menu.

You can search for the file currently displayed in the Editor by selecting **Find** and **Find Next** from the **Edit** menu.

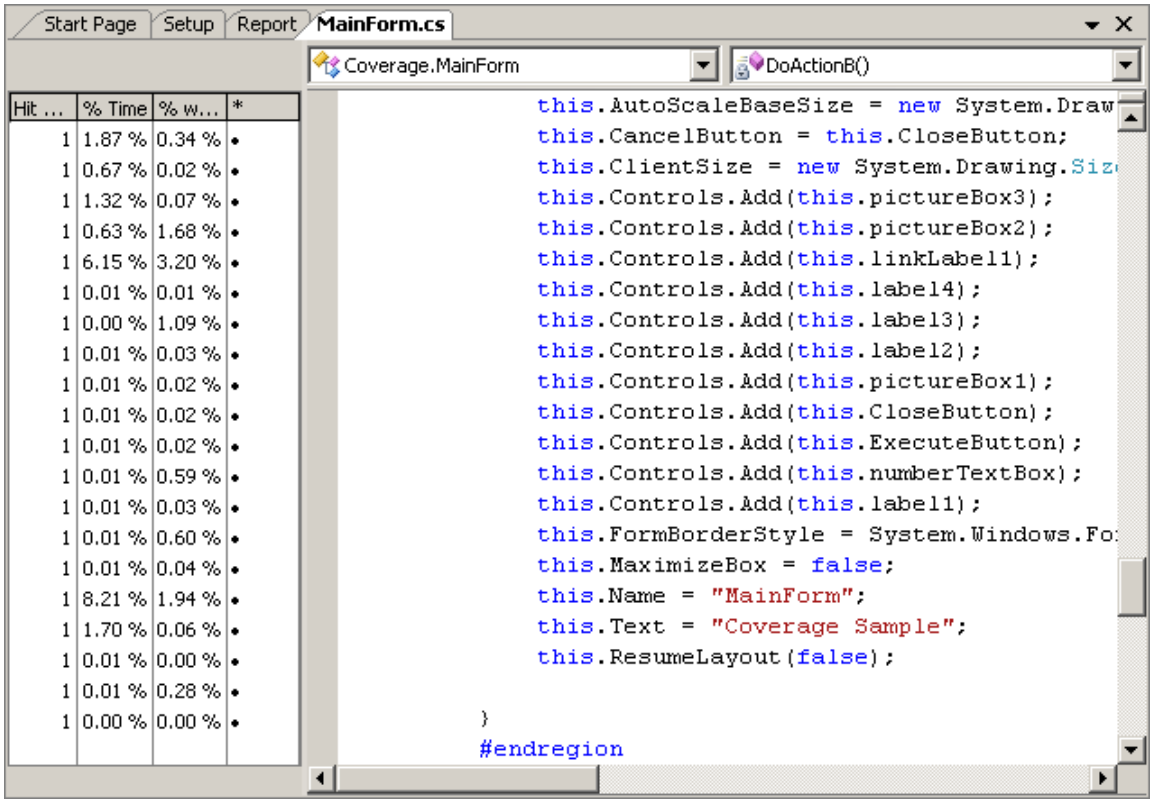
To print the file that is displayed in the Editor, simply select **File | Print** from AQtime's main menu. Another item of the File menu, **Print Preview**, will open the Print Preview form instead of sending the file directly to the printer. Using this form, you can select the paper size, specify margins, background, etc. The profiling results for routines are printed as comments before the routine's code. The profiling results for lines

(those that were displayed in the gutter) are also transformed into comments that are placed before each line of code.

The **Editor** panel displays the source code in AQtime running as a standalone application. If AQtime is integrated into Visual Studio, AQtime's Editor panel is not available. The source code is displayed in the **Code Editor**, which is Visual Studio's native text and code editor. AQtime extends its functionality to provide interaction with AQtime panels and to display the profiling results along with the source code.

AQtime updates the Code Editor to display the source for the last routine double-clicked in the **Setup**, **Summary**, **Report**, **Details**, **Call Graph**, **Call Tree, PE Reader** or **Event View** panel, unless the routine's source file cannot be found on the search path (an alternative way to display the source code for the routine currently chosen in any of the aforementioned panels is to select **Edit | Show Source File** from Visual Studio's main menu). The application sources may not be displayed in the following cases:

- Your executable was not compiled with debug information: both managed and unmanaged modules must be compiled with debug information in order for AQtime to be able to find information on the routine's source file. See *How AQtime Profilers Use Metadata and Debug Information*. To solve the problem, please compile your application with debug information.
- The debug info file is absent. This may happen, for instance, if you compiled your application with debug information, but added another file (it has the same name, but does not include debug information) to your AQtime project. Please make certain that you included the appropriate module.
- The source file is not displayed when you select a class in the Report, Details or another AQtime panel. This happens because debug information does not hold information about source files for classes.
- The source file has been removed or it has never been created (some compilers, for instance, Visual Basic .NET, can compile the executable without saving its sources to a disk file).
- The source file cannot be found in the search path. To specify the search path, use the **Project Search Directories** or **Search Directories** dialog (to call them, select **Project | Search Directories** or **Profile | Options | Search Directories** from Visual Studio's main menu).



To search for or replace text in the Code Editor, use the standard Visual Studio methods.

To view profiling results of a routine whose source code is currently displayed in the Code Editor, in the Report panel, right-click somewhere in the routine's code and select **Synchronize Report** from the context menu.

AQtime adds a grid to the Code Editor and displays results of line profiling in this grid (see the image above). The grid holds the same set of columns as the **Lines** table of the **Details** panel holds plus one extra column: *. This column holds bullets for lines that were profiled. If you click a bullet, AQtime will show the **Routine Summary** dialog displaying profiler results for a routine that holds the line which bullet you have clicked. An alternative way to view routine results is to right-click somewhere within the source code of the desired routine and select **Show Routine Summary** from the context menu (this will also open the Routine Summary dialog).

Also you can view profiling results without opening the Routine Summary dialog. To do this, hold the mouse pointer over the bullet until the hint is shown. If a routine was profiled at line level, the hint will display the profiling results for the line. If a routine was profiled at routine level, the hint will show the routine results.

Note that grid columns show profiling results for routines that were profiled at line level. If a routine was profiled at routine level, the grid columns will be empty, except for the * column. You can click bullets in this column to view profiling results of your routines.

You can collapse and expand blocks of code in the Code Editor. However, the grid is not updated when you collapse or expand blocks. This happens because Visual Studio does not notify AQtime about collapsing and expanding. So, to ensure that the grid shows appropriate profiling results for routine and source lines, expand all the blocks. You can do this using the **Outlining | Toggle All Outlining** or **Outlining | Stop Outlining** item of the Code Editor's context menu.

© 2010 AutomatedQA Corp.

Like in other AQtime panels that contain grids, you can move the columns within this grid as well as add columns to or hide them from the grid. See *Arranging Columns, Lines and Panels* for details. In addition, you can configure the format of the grid columns. To do this, use the **Format Columns** dialog, which is called by selecting **Format Columns** from the context menu.

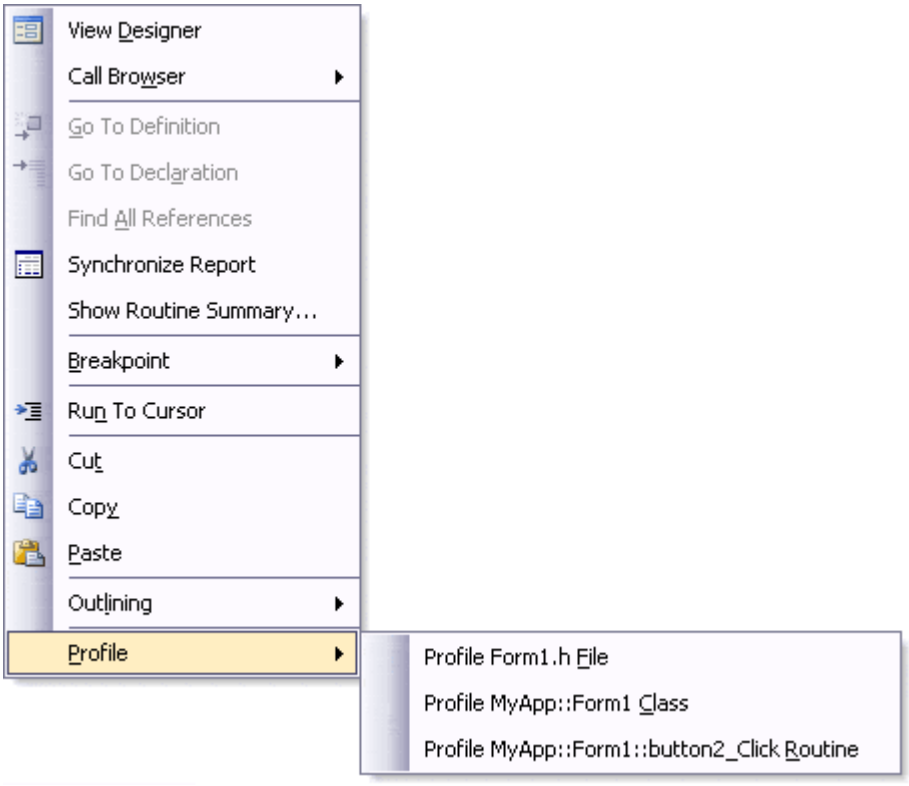
The Code Editor's context menu holds two items for saving and restoring the grid appearance. The **Save to Default View** item stores the grid settings to the *Default* result view. **Restore Default View** applies the grid settings with settings stored to the *Default* result view.

The grid footer displays summary values for selected grid rows. If there are no rows selected, the footer shows a summary for all rows.

If you split the Code Editor, the grid will be displayed in both parts of the Code Editor's window.

To hide the grid, select **Hide** from its context menu. The grid will become visible automatically when you double-click a routine in one of AQtime panels (**Report**, **Details**, **Call Graph**, etc.)

Also, you can select which elements to profile via the **Profile** submenu of the Code Editor's context menu.



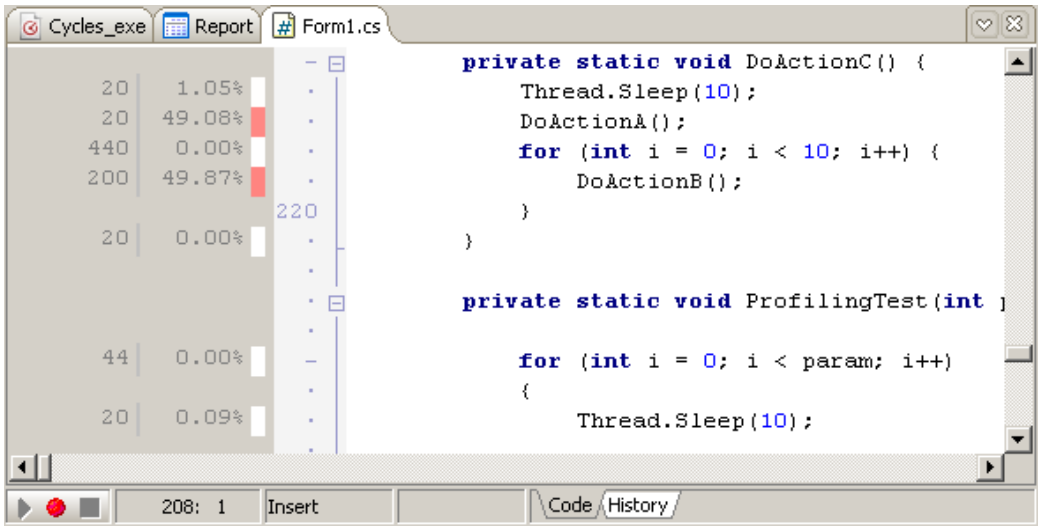
This menu allows you to profile the routine where the cursor is currently located in the code, the class where the cursor is located or the entire edited source file. Selecting the corresponding submenu item will add the specified routine, class or source file to an including area and start profiling.

Let repeat it again that the **Editor** panel displays source code in AQtime when it is running as a standalone application. If AQtime is integrated into Borland Developer Studio, AQtime's Editor panel is not available. The source code is displayed in Borland Developer Studio's native text and code editor. AQtime extends its functionality to provide interaction with AQtime panels and to display the profiling results along with the source code.

AQtime updates the **Editor** to display the source for the last routine double-clicked in the **Setup**, **Summary**, **Report**, **Details**, **Call Graph**, **Call Tree**, **PE Reader** or **Event View** panel, unless the routine's

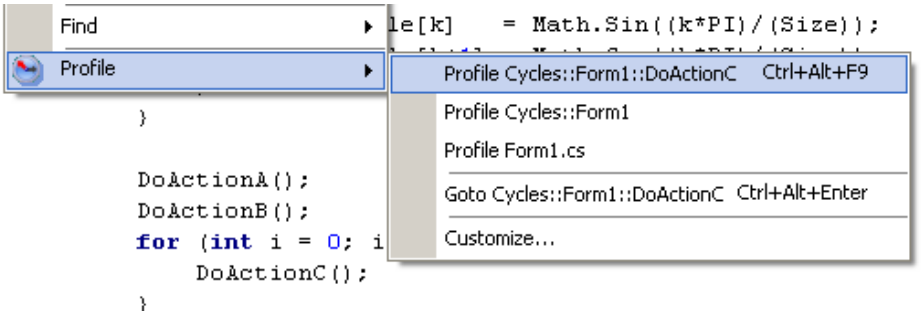
source file cannot be found in the search path. The application sources may not be displayed in the following cases:

- Your executable was not compiled with debug information: both managed and unmanaged modules must be compiled with debug information in order for AQtime to be able to find information on the routine's source file. See *How AQtime Profilers Use Metadata and Debug Information*. To solve the problem, please compile your application with debug information.
- The debug info file is absent. This may happen, for instance, if you compiled your application with debug information, but added another file (it has the same name, but does not include debug information) to your AQtime project. Please make sure that you included the appropriate module.
- The source file is not displayed when you select a class in the Report, Details or another AQtime panel. This happens when debug information does not contain information about source files for classes.
- The source file has been removed or it has never been created.
- The source file cannot be found in the search path. To specify the search path, use the Project Search Directories or Search Directories dialog.



To search for or replace text in the Editor, use the standard Borland Developer Studio methods.

If you want the Report panel to display profiling results of a routine whose source code is currently displayed in the Editor, do the following: right-click somewhere within the routine's code and select **Goto <Routine_Name>** from the **Profile** submenu of the context menu. This will update not only the Report panel, but also all the other panels of AQtime that are related to Report. Additionally, the Profile submenu lets you profile the routine in whose code the cursor is currently located, the class where the cursor is located or the whole active source file. Selecting the corresponding item in the Profile submenu will add the specified routine, class or source file to an including area and start profiling.



AQtime adds a grid to the Editor and displays results of the profiling in this grid. If a routine was profiled at line level, the grid can display the same set of columns as the **Lines** table of the Details panel. If a routine was profiled at routine level, the grid holds the same set of columns as the Report. If you place the mouse pointer over the grid, you will see the profiling results of the selected routine (routine line) as a hint. If the routine was profiled at line level, the hint will display the line profiling results. If the routine was profiled at line level, the hint that is called on any line of the routine will display the routine profiling results.

Besides routine or line level profiling results, the grid displays the indicator that shows which routine or source code line is problematic. The indicator is a rectangle whose saturation of red depends on the alert level: white means the routine or line is healthy, pink means you should pay attention to it, deep red means that this routine or line causes a performance problem.

You can collapse and expand blocks of code in the Editor. However, the grid is not updated when you collapse or expand blocks. This happens because Borland Developer Studio does not notify AQtime about collapsing and expanding. So, to ensure that the grid shows appropriate profiling results for routine and source lines, expand all the blocks. You can do this using the **Unfold | All** item of the Editor's context menu.

To configure which of the available columns to display in the grid and which of them to hide, right-click the routine's source code and select **Profile | Customize** from the context menu. This will call the **Customize** dialog, where you can choose the desired columns and uncheck the unwanted ones.

Editor Panel – General Settings

The **General** group of the Editor's options contains one setting:

- **Keep highlighting when printing** - If this option is enabled, the Editor panel follows active color settings when printing source files. For instance, if you use white letters on a black background, the Editor will print white letters on a black background. If the option is disabled, the Editor only uses font style highlighting for printing (bold, italic and so on) and does not use color highlighting (that is, the font color is always black and the background is always white).

Editor Panel – Display Settings

Settings of the **Display** group of Editor's options control how Editor displays source code files. The Display settings are organized into the following sections:

Text

- **Font name** - This list box displays the installed fonts that support the chosen *Font character set*.
- **Font size** - Font size in points.
- **Font character set** - Specifies which font character set to use. This option affects how the upper 128 ANSI characters will be displayed.
- **Control characters** - If this option is enabled, Editor displays spaces, tabs and line breaks using special characters, so that you can distinguish, for instance, a tab from a string of spaces.

Background

- **Style** - Specifies the background style. You can select one of the four gradient effects or empty background (*None*). To select the color of the gradient fill, use the **Start color** and **End color** controls.

- **Start color** and **End color** - Specifies the start and end color for the gradient fill and are used when the *Style* option is not *None*.
- **Right margin** - The right margin is a thin vertical line to specify where your break lines are. This setting defines whether the line is shown in the panel.
- **Right margin position** - Specifies the position of the right margin line within the panel. The position is specified in pixels from the left side of the Editor window.

Gutter

The gutter is the gray area to the left of the Editor panel where markers and line numbers are displayed. These settings allow you to specify the visual style of the gutter.

- **Style** - Specifies the background style for the gutter. You can select one of the four gradient effects or *None*. To select the color of the gradient fill, use the **Start color** and **End color** controls.
- **Start color** and **End color** - Specify the start and end color for the gradient fill and are used when the *Style* option is not *None*.
- **Gutter capacity** - Specifies the gutter width in relative units. It means how many markers can be placed in the gutter on one line.
- **Line numbers** - Specifies whether line numbers will be displayed in the gutter.

Editor Panel – Highlighting Settings

The **Highlighting** group of the Editor's settings specifies the highlighting format to be used to display different syntax elements in the Editor.

The list of syntax elements is shown on the the left of the dialog. There are several settings for each syntax element. Some of the elements, for example, selected text, have fewer options than other elements. The entire set is as follows:

- **Colors** - Specifies the foreground and background colors that are used to display syntax elements.
- **Text attributes** - Font style (bold, italic, underlined and strikeout -- **B**, *I*, U and ~~S~~).

As you change the syntax elements' settings, the Editor panel automatically applies these changes, so that you can preview them directly in the Editor.

You can either define the highlighting settings yourself, or load the settings that are used in Visual Studio or Delphi:

- To import Visual Studio highlighting settings, press **Import Visual Studio Highlighting**. This feature supports Visual Studio version 7.x, 2005 and 2008. If you have several versions installed on your computer, the settings of the latest available version will be imported.
- To import the highlighting settings of Borland Developer Studio or CodeGear RAD Studio, press **Import BDS Highlighting**. This feature requires Embarcadero RAD Studio 2010, CodeGear RAD Studio 2007 or 2009, or Borland Developer Studio 2006 to be installed on your computer.
- To import Delphi highlighting settings, press **Import Delphi Highlighting**. This feature requires Borland Delphi 7 to be installed on your computer.

To restore the default highlighting settings, press **Restore Default Highlighting**.

AQtime supports various development tools that use different programming languages. These languages have different keywords, different syntax, rules for comments and so on. In the table that is displayed at the bottom of the page, you can specify which highlighting rules the Editor will use to display these types of files. The value shown in the **Language** column is the parser that AQtime will use to highlight source code in the

Editor panel. The **Extensions** column specifies the files' extensions that the parser will work with. For instance, the following row –

Language	Extension
C#	*.cs; *.scs

-- means that the Editor will treat .cs and .scs files as C# files.

To modify a file type, click the desired Extensions cell and type the desired value. Specify the file types with the asterisk and the leading point. Use semicolons to separate multiple types.

Editor Panel – User Keywords Settings

The **User Keywords** dialog lets you define words that the Editor panel will highlight in a special style. This is the style that is specified for the **User reserved word** element in the Highlighting Settings dialog.

Note: The style that is set for the **User reserved word** element overrides any other styles that may be applied to a word you added to user keywords.

You can separate individual keywords in the list with any non-alphanumeric symbols, including line-breaks.

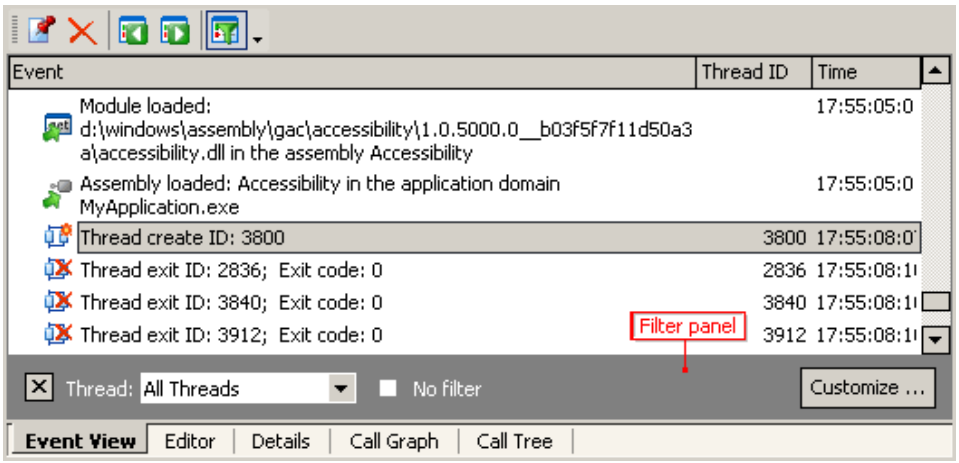
Event View Panel

Event View Panel - Description

The **Event View** panel displays events that occur within AQtime and the profiled application during profiling. Each event is displayed as a parent node. Some events have child nodes that provide you with additional information about the event.

To display the Event View panel, do any of the following:

- *AQtime standalone:*
  - Select **Event View** from the **View | Other Panels** menu.
  - Select **Event View** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.
  - Select **Event View** from the **Assistant** panel.
- *AQtime integrated into Visual Studio:*
  - Select **Event View** from the **Select Panel** dialog, which is called by selecting **Profile | Panel List**.
  - Select **Event View** from Visual Studio's **Solution Explorer**.
  - Select **Event View** from the **Assistant** panel.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Event View** from the **View | Profile Windows | Other** menu.
  - Select **Event View** from the **Assistant** panel.



There are tree columns: **Event** holds the event description, **Thread ID** holds the identifier of the thread where the given event occurred, **Time** holds the time of occurrence. If the *Time from application start* option is enabled, Time is counted from the beginning of the current profile run. Otherwise, it is the system time. You can arrange the Event View panel the same way you can organize other AQtime panels.

At any time you can stop or start monitoring events by unchecking (or checking) Active from the panel's context menu.

To search for text in the Event column, either select **Find** and **Find Next** from the **Edit** menu (this will call the Find dialog that is used for searching), or simply focus the column and type the desired text.

To restrict the type of events displayed (see the list below), select **Filter Panel** from the context menu, which will display the **Filter** panel at the bottom of the Event View panel. Pressing *Customize* will bring up the **Message Filter** dialog, in which you can exclude unnecessary event types by unchecking them. The filter is active only if the checkbox in the Filter panel is enabled. Correspondingly, if you want to disable the filter temporarily, disable the checkbox. In addition, you can filter messages in the Event View panel by the thread for which they were logged. For this purpose, simply select the desired thread in the **Thread** dropdown list box that is displayed within the Filter panel. *All Threads* means that Event View will display messages for all threads, while *Null Thread* stands for messages whose Thread ID is empty.

To specify the font color for each type of event displayed, use the **Message Colors** dialog, which is called by selecting **Message Colors** from the context menu.

To copy the selected events to the clipboard, choose **Copy** from the **Edit** menu. The context menu also holds **Add Comment**, which lets you add comments as "events" into the event list. To paste the contents of the clipboard to the event list, select **Paste as Comment** from the context menu. To edit the chosen custom comment, simply double-click it.

To export the selected events or all of the events in the panel to an HTML, XML or TXT file, choose **Save Selection** or **Save All** from the context menu. You can also use the Ctrl and Shift commands to multi-select events. (See *Selecting Several Records in a Panel*.)

The displayed event types can be divided onto two groups:

- **Standard events**
- **.NET events**

To browse through events of the same type, use the **Go to Next Event** and **Go to Previous Event** buttons on the Event View toolbar. Using the and toolbar buttons, you can browse through events that were logged for the same thread.

Two notes:

- When you profile an ASP.NET application, all General Events displayed in the Event View panel come from the ASP.NET process while .NET Specific Events come from the profiled ASP.NET application. **ASP.NET process** is the *aspnet_wp.exe* or *w3wp.exe* (if you work with IIS 6.0) executable. It is located in the *<Windows>/Microsoft.NET/Framework/<Framework Version>* folder.
- Your application can post messages to the Event View panel. To do this, simply call the Windows API `OutputDebugString(Str)` function, that is defined in Kernel32.dll. AQtime traces calls to this function and posts the string, which is passed to the function as a parameter, to Event View. The string is logged as the Debug String event.

Standard Events

Event	Description
Attach to Process	Logs the attaching of the current profiler to a process.
Change Profiler	Logs a change of profilers.
Comment	This is the equivalent of a String Received event, but for a comment added by the user via the Add Comment dialog.
Create Process	Logs creation of the profiled application's process and gives its process ID, handle and base address, as well as the ID and handle of its primary thread.
Create Thread	Logs creation of a secondary thread in the profiled application and gives its thread ID and handle.
Debug String	Logs calls to the Windows `OutputDebugString` function, with the string passed as a parameter to the function. For more information on using this function, see *Adding Custom Messages to the Event View Panel*. AQtime tracks the call stack for the `OutputDebugString` function and displays call stack entries as child nodes of the Debug String node (see description of the Stack Entry event).
Debug Symbols Read	Logs the end of the debug-info reading process at the start of a profile run.
Detach From Process	Logs the detaching of the current profiler from a process.
Exception	Logs system exceptions raised by the profiled application, with the exception code, name and address, and the call stack, displayed as child nodes (see *Exceptions in the Event View Panel*). The panel's *Max consecutive exceptions* option defines the point beyond which it will stop logging exceptions until the next non-exception event. AQtime tracks the exception's call stack and shows it as child nodes of the exception node (see description of the Stack Entry event). Note that this event is used to log information about system exceptions. .NET exceptions are logged with special *.NET Exception* event (see below).
Exit Process	Logs the closing of the profiled application's process, with process ID and exit code.

Exit Thread	Logs the closing of a secondary thread in the profiled application and gives its thread ID, handle and exit code.
Failed to Start the Process	Logs the failed attempt to start profiling.
Module Activated	Indicates that the project's "main" module was changed in the **Setup** panel.
Module Added	Logs the addition of a module (exe, dll, etc.) to the current AQtime project.
Module Loaded	Logs the loading of a module (dll, external executable, etc.) by the main application and gives its base address in memory once loaded and its size in bytes.
Module Removed	Logs the removing of a module from the current AQtime project.
Module Unloaded	Logs the release of a module by the main application and gives its base address before unloading.
Process Resumed	Logs the moment a profile run is resumed after being suspended from the AQtime interface.
Process Suspended	Logs the moment a profile run is suspended from the AQtime interface.
Profiling Disabled	Logs the moment the profiling status is disabled upon selecting the **Enable/Disable Profiling** item from AQtime's **Standard** toolbar, Borland Developer Studio's **Run.AQtime** toolbar or Visual Studio's **Profile** menu.
Profiling Enabled	Logs the moment the profiling status is enabled upon selecting the **Enable/Disable Profiling** item from AQtime's **Standard** toolbar, Borland Developer Studio's **Run.AQtime** toolbar or Visual Studio's **Profile** menu.
Project Closed	Logs the closing of a project in AQtime.
Project Loaded	Logs the opening of a project in AQtime.
Project Run	Logs the start of a profile run. This message also reports the command-line arguments, host application and working folder used for the run.
Results Generated	Logs the end of result generation after a profile run, and displays a summary of the results.
Stack Entry	These items show information about routines in the exception, debug string or user breakpoint call stack. For each routine, the address, module name, routine name, etc. are given. If there is enough information to retrieve them, the best substitute is displayed, for instance the routine address in memory when the name is unavailable (for information on how to solve this problem, see *Event View Panel - Possible Problems With the Call Stack*). Double-clicking on any routine in the stack will update the **Editor** panel to its source code.
User Breakpoint	Logs execution of the int3 assembler instruction. This instruction is used for debugging purposes very often. For instance, it is used to implement assertions and check memory overwrites. AQtime traces

	the call stack for `int3` and displays call stack entries as child nodes of the User Breakpoint node.
String Received	Logs and shows a string message generated by AQtime or one of its profilers or plug-ins for the event list.
Work Environment	Logs environment information on the current computer.

.NET Events

Event	Description
Application Domain Created	Logs creation of an application domain with the application domain ID. This ID is valid for any information request after the application domain has been fully created.
Assembly Loaded	Logs the loading of an assembly and gives the assembly ID.
Module Loaded	Logs the loading of a module and gives the module ID.
.NET Exception	Logs a .NET exception that occurred in the profiled managed code with the exception's class name. AQtime tracks the exception's call stack and shows it as child nodes of the exception node (see description of the Stack Entry event). Note that for each .NET exception the CLR generates a system exception that is also traced by AQtime and logged to the Event View panel.

Event View Panel Options

The Event View panel offers several customizable options. You can check or change these options using the **Event View Options** dialog. To call it, do any of the following:

- *AQtime standalone:*
  - Select **Options** from the context menu of the Event View panel.
  - Select **Options | Options** from AQtime's main menu. This will call the Options dialog. Event View Options belong to the **Profiling Time** group.
- *AQtime integrated into Visual Studio:*
  - Select **Options** from the context menu of the Event View panel.
  - Right-click **Event View** in Visual Studio's **Solution Explorer** and select **Properties** from the context menu.
  - Select **Tools | Options** from the main menu of Visual Studio and then select the **AQtime | Profiling Time | Event View** group in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Options** from the context menu of the Event View panel.
  - Select **Profile | Options** from the main menu of Borland Developer Studio. This will call the Options dialog. Event View Options belong to the **Profiling Time** group.

All options are divided into three large groups, **General**, **Display Events in** and **Exception filter** which you select on the left of the dialog:

General

This group holds options that specify how Event View handles events.

- **Clear on application start** – Sets whether Event View will begin empty each time the profiled application is launched, that is, on each profile run. Else, it will keep events from previous runs.
- **Time from application start** – If enabled, times shown in the **Time** column are elapsed times from application start. Else, they are system time.
- **Show stack address as RVA** - This option specifies what format should be used to display stack addresses in events and messages. The address of each routine consists of two components: The base address of the module, which is the address where the module is loaded in memory, and the offset of the routine relative to this base address. The offset is also called a *relative virtual address* (RVA). If *Show stack address as RVA* is enabled, the event messages hold only the relative virtual addresses. Otherwise, they hold the full routine addresses, that is, the base address + the offset. Note that the base address can only be determined after the module has been loaded into memory. The preferred loading address is used as the base one. The preferred loading address is specified in the header of the executable. To find it in AQtime, check the **Optional header** section on the PE Information page of the PE Reader panel.
- **Generate dump on exception** - If this setting is enabled, AQtime automatically generates a dump file when it detects an exception in the profiled application. The generated dump file contains the information that help you find the cause of the exception: memory data, the exception code, and address, information about application's threads and loaded modules and so on.

  The dumps are saved to the folder specified by the *Dump folder* setting (see below).

  Note: The dumps are not generated for those exceptions that are filtered out and not logged to the Event View panel.

  For detailed information on automatic dump generation, see *Generating Dumps for Profiled Applications*.
- **Dump folder** - Specifies the folder, to which AQtime will automatically save dumps when it detects an exception in the profiled application (see description of the *Generate dump on exception* setting above). Note that the folder must exist; AQtime does not create it automatically.
- **Debug Events** - Settings for the display on the Debug String events. None of them has effect unless you enable *Show call stack*.
  - **Show all parents** - The call stack may include functions for which there is no debug info. Typically these functions are located in pre-compiled libraries. If this option is enabled, the call stack will include these functions. Otherwise, they will be suppressed from the call stack display.
  - **Show full module path** - If it is enabled, the full path to modules is displayed in stack frames.
  - **Show call stack** - This section contains options that specify whether AQtime traces the stack of function calls for the following debug events:
    - Module Loaded
    - Module Unloaded
    - Debug String
    - User Breakpoint
    - Process Suspended

    The **Depth shown** setting specifies the number of routines in the call stack. Default: 10. 0 means no tracing.

© 2010 AutomatedQA Corp.

- **Exceptions** - Settings for the display of exception events. None (except *Active*) has any effect unless *Active* is enabled.
  - **Active** - Enables exception logging. When enabled, exceptions are shown in Event View, as set by the sub-options below, and their time is counted in the function where they occur. Else, exceptions are neither logged as events nor counted as part of execution time.
  - **Hide IsBadPtr exceptions** - This option affects applications that use the `IsBadReadPtr`, `IsBadWritePtr`, `IsBadCodePtr` or `IsBadStringPtr` Windows API functions. If the option is enabled, exceptions raised by these functions will not be logged to Event View. See *Exceptions in the Event View Panel*.
  - **Depth shown** - If *Show call stack* is on, this option specifies the number of traced routines in the stack. Default value: 10. 0 means no tracing.
  - **Show all parents** - The call stack may well include functions for which there is no debug info, typically from pre-compiled libraries. If this option is enabled, the call stack shown for exception events will include these functions. Else, they will be suppressed from the call stack display.
  - **Max consecutive exceptions** – Number of exception events, uninterrupted by any other event, after which exception logging will be disabled until the next non-exception event. This saves profiling time during exception loops.
  - **Show full module path** - If it is enabled, the full path to modules is displayed in stack frames when an exception occurs.

Display events in

This group includes options that specify where AQtime outputs event and message flow.

- **Event View panel**
  - **Active** - If this option is enabled, AQtime outputs events and messages to the Event View panel.
- **NT event log**
  - **Active** - If this option is enabled, AQtime outputs events and messages to the Application Log section of the Event Viewer of Windows NT, 2000, XP or 2003. To see the error log, open **Control Panel | Administrative Tools | Event Viewer** in Windows 2000, XP or 2003, or **Control Panel | Event Viewer** in Windows NT.
  - **Event source name** - Specifies the string that will be displayed in the **Source** column of the Event Viewer.
- **Text file**
  - **Active** - If this option is enabled, AQtime writes the event flow to a text file.
  - **File name** - The name of the text file for output.
- **XML file**
  - **Active** - If this option is enabled, AQtime writes events and messages to an .xml file.
  - **File Name** - The name of the .xml file for output.

Exceptions filter

The settings in this group specify the exceptions to be excluded from the Event View's panel. This functionality helps you exclude unimportant issues from the analysis and concentrate your efforts on critical exceptions.

The **Exceptions filter** page contains a list of Win32 exceptions to be excluded from analysis. The Event View panel will skip those exceptions that are added to the list and that are selected in this list. If an exception is not selected (its check box is clear) or if it is absent from the list, the Event View panel will log this exception.

By default, the list contains Win32 exceptions. You can add them to or exclude them from the filter by selecting or clearing the appropriate check box.

To add a new exception to the list:

- Click **Add**. This will invoke the Add Exception dialog.
- In the dialog:
  - Enter the code of the desired exception into the **Code** edit box.
  - In the **Description** box type any descriptive text.
  - Press **OK**.

The exception will be added to the list.

To remove an exception from the list:

- Select the desired exception in the list.
- Click **Delete**.

Exceptions in the Event View Panel

The Event View panel logs information about exceptions and displays the sequence of function calls that caused the exception. This topic describes some peculiarities of tracing exceptions.

Controlling IsBadPtr Exceptions

AQtime must launch the application being profiled as a child process of AQtime itself, while in debugging mode. The profiled application may call Windows API functions. Some of these functions change their behavior under AQtime. For instance, in normal non-debugging mode, API functions `IsBadCodePtr`, `IsBadReadPtr`, `IsBadWritePtr` and `IsBadStringPtr` return a non-zero value if invalid memory addresses are passed to them. In debugging mode the same invalid addresses will raise exceptions, which will of course be displayed in the **Event View** panel.

To suppress the display of these debugging-mode exceptions, enable the *Exceptions* | *Hide IsBadPtr exceptions* option of the Event View panel.

Controlling Consecutive Exceptions

The number of consecutive, uninterrupted, exceptions that will be logged to Event View is set by the *Exceptions* | *Max consecutive exceptions* option. When consecutive exceptions overflow this limit, Event View stops tracing them until it gets at least one non-exception event.

System and .NET Exceptions

Exceptions that occur in unmanaged (native) code are logged with the **Exception** events. Exceptions that occur in managed code are logged with the **.NET Exception** event. For each .NET exception the CLR generates a system exception that is traced and logged by AQtime.

Note: AQtime does not trace exceptions that occur in console applications created for .NET Framework ver. 1.0 and 1.1.

© 2010 AutomatedQA Corp.

Generating Dumps on Exceptions

You may command AQtime to generate dumps on exceptions automatically. To do this, enable the *Generate dump on exception* setting of the Event View panel and specify the folder that AQtime will save the generated dumps in, in the *Dump folder* setting.

If an exception occurs, AQtime checks the state of the *Generate dump on exception* setting and if this setting is enabled, AQtime generates a dump and saves it to the folder, specified by the *Dump folder* setting. For detailed information, see *Generating Dumps for Profiled Applications*.

Two notes:

- AQtime does not generate dumps for those exceptions that are filtered out and not logged in the Event View panel (see below).
- When a .NET exception occurs, the CLR also generates a system exception. AQtime will export this system exception, not the .NET exception. The exception' call stack will contain native-code information. It will not contain the names of managed routines.

Filtering Exceptions

By using the options of the Event View panel you can filter Win32 exceptions to be logged. The panel will only display those exceptions that are not included into the filter. This feature lets you ignore the exceptions that are not important and concentrate on critical issues.

To specify the exceptions to be skipped, use the *Exceptions filter* settings:

- Select **Options | Options** from AQtime's main menu to display the Options dialog.
- From the list of settings on the left of the dialog, select the **Profiling Time | Event View | Exceptions filter** group. On the right, AQtime will display the list of exceptions to be filtered. By default, the list contains Win32 exceptions.
- To command AQtime to skip certain exceptions when profiling your applications, select the appropriate check boxes in the list and press **OK** to save changes.

We would like to note that AQtime will only skip those exceptions that are added to and selected in the list. If an exception is not selected or it absents from the list, it will be displayed in the Event View panel.

To add an exception to the list, press **Add** and then specify the exception's code and description in the ensuing Add Exception dialog.

Adding Custom Messages to the Event View Panel

When you are profiling your application with AQtime, the **Event View** panel displays events that occur in the profiled application and in the operating system during profiling. The panel shows the event log at runtime and after the application terminates. You can find a list of traced events in the *Event View Panel* topic.

You can add custom messages to the event log manually and programmatically. It is easy to do this. To add a message to the Event View panel manually:

- Switch to the **Event View** panel,
- Select **Add Comment** form the panel's context menu and
- Specify your messages in the subsequent **Add Comment** dialog.

or

- Copy the message text to the clipboard.
- Switch to the **Event View** panel and

- Select **Paste as Comment** from the panel's context menu.

The new messages will be logged as comments at the end of the event log or after the currently selected message.

To modify the added comment, double-click it in the Event View panel and then change the comment in the subsequent Edit Comment dialog.

To add a message to the Event View panel programmatically, call the `OutputDebugString` function within your code. `OutputDebugString` is a Windows API function that sends a string to the debugger. Since AQtime acts as a debugger for your application, it receives the string and displays it in the Event View panel as the Debug String event.

Normally, the number of various events that occur during the profiler run is large. Using the `OutputDebugString` function, you can easily insert "markers" into the event log to indicate that certain operation has been started (or finished) or to signal some changes in the application. All you have to do to insert a marker is to call the `OutputDebugString` function in your code.

The Event View panel can determine the hierarchy of function calls that led to the call to the `OutputDebugString` function. To enable the call stack tracing, switch on the *Common* | *Debug Strings* | *Show call stack* option in the **Event View Options** dialog (to call this dialog, select **Options** from the Event View context menu).

The call stack is shown as child nodes of the Debug String node. These child nodes hold the addresses and names of routines that are in the call stack. For more information on possible problems with call stacks, see *Event View Panel - Possible Problems With the Call Stack*.

Event View Panel – Possible Problems With the Call Stack

When you are profiling your application with AQtime, the **Event View** panel displays events that occur in the profiled application and in the operating system during the profiling. You can find a list of traced events in the *Event View Panel* topic.

For two events - *Exception* and *Debug String* - the Event View panel shows the call stack:

- The *Exception* event indicates that an exception occurs in the profiled application. The call stack shows the hierarchy of function calls that led to the exception.
- The *Debug String* event logs a call to the Windows `OutputDebugString` function. The call stack of this event displays the hierarchy of function calls that led to the call of that function.

The call stack is shown as child nodes of the Exception or Debug String nodes. The call stack nodes hold the addresses and names of routines that are in the call stack.

When viewing the call stack that you may come across the following problem: the call stack nodes hold only routine addresses, but not the routine names. This happens because AQtime cannot find the names for those routines in debug information. That is, the call stack shows the name of a routine only if AQtime can find debug information for that routine. You can fix the problem by compiling these modules with debug information. For more information on this, see *How AQtime Profilers Use Metadata and Debug Information*.

Stack Frames – Compiler Settings

This topic describes compiler options that specify whether your application will be compiled with or without stack frames:

Microsoft Visual C++ .NET

- Open the **Project Properties** dialog.

- Select the **Configuration | C/C++ | Optimization** category from the tree list on the left.
- On the right part of the dialog, assign **No** to the **Omit Frame Pointer** setting in order to compile your application with stack frames. To compile the application without stack frames, set the option to **Yes**.

Microsoft Visual C++ 6.0

- Open the **Project Settings** dialog.
- Switch to the **C/C++ tabbed** page.
- Select **Optimizations** from the **Category** box.
- Select **Customize** from the **Optimizations** box. This will enable the list box that is displayed below the **Optimizations** box.
- Find the **Frame-Pointer Omission** option in the list box.
- Enable this option to compile your application without stack frames. To compile the application with stack frames, uncheck the option.

Borland Delphi

- Open the **Project Options** dialog.
- Switch to the **Compiler** page.
- To compile your application with stack frames, check the **Stack frames** box in the **Code Generation** section. Uncheck this box to compile your application without stack frames.

Borland C++Builder

- Open the **Project Options** dialog.
- Switch to the **Compiler** page.
- To compile your application with stack frames, check the **Stack frames** box in the **Compiling** section. Uncheck this box to compile your application without stack frames.

Explorer Panel

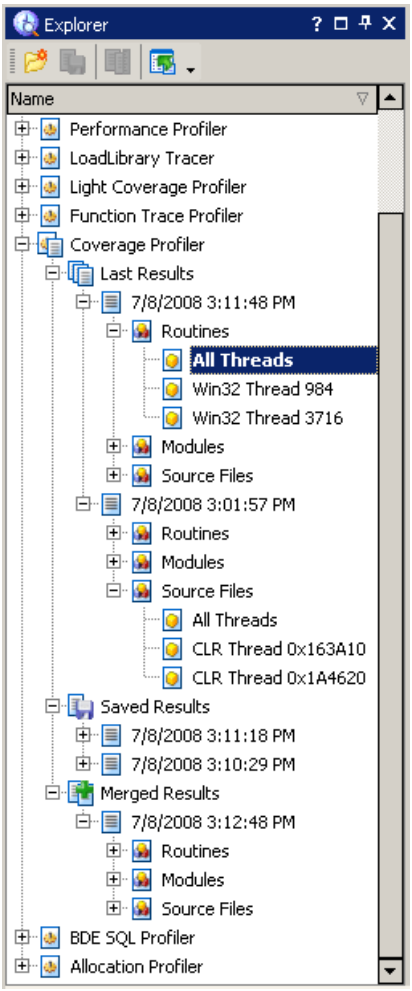
Explorer Panel - Description

AQtime's **Explorer** panel serves to manage the profiling results. It supports the following operations:

- Save the current profiler results for future use.
- Load previously saved results and display them in the **Report** panel.
- Delete previously saved results when they are no longer of use.
- Compare two or more result sets.
- Export results to a binary file.
- Import results from a binary file.
- Collect related result sets and organize them into folders on the Windows Explorer model.

To display the Explorer panel, do any of the following:

- *AQtime standalone:*
  - Select **Explorer** from the **View** menu.
  - Select **Explorer** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.
  - Select **Explorer** from the **Assistant** panel.
- *AQtime integrated into Visual Studio:*
  - Select **Explorer** from the **Select Panel** dialog, which is called by selecting **Profile | Panel List**.
  - Select **Explorer** from Visual Studio's **Solution Explorer**.
  - Select **Explorer** from the **Assistant** panel.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Explorer** from the **View | Profile Windows** menu.
  - Select **Explorer** from the **Assistant** panel.



The organization of this display can be modified in the same way as for other AQtime panels. If Explorer's *Show results for all profilers* option is disabled (the default, shown above), it only displays results for the currently selected profiler. Else it displays a tree with a branch for each profiler, and in each a sub-tree of folders identical to the tree shown above.

© 2010 AutomatedQA Corp.

All items inside folders are names for result sets, and they can be edited in place. The default name is simply the date and time that the results were generated. In addition, depending on the current profiler, each result set is divided into one or more result profiles (for instance, for the **Allocation** profiler there are **Classes** and **Objects** profiles, for **Performance** – **Routines, Source Files** and **Modules**). A profile specifies the kind of items in the results table (routines, classes, objects, etc.) For multithreaded applications, separate results per thread are kept as sub-items of the whole-application results in the selected profile. Double-clicking on the profile name or on the **All Threads** node will open the thread-results list.

There are a number of main branches to the results tree. Individual result sets can be dragged from one to the other, or click-dragged to copy them. They can be deleted by using the Del key or **Delete** on the context menu. Or they can all be deleted by using **Clear All**. Generally, the same manipulations are possible in the Explorer tree as in Windows Explorer, but some are forbidden for obvious reasons -- you cannot drag or copy into **Last Results**, for instance.

The folders are:

- **Last Results** – The most recent results are automatically kept, up to a certain number, with each new result set expelling the oldest from the list. The number of result sets kept here is set by the *Number of recent results to keep* Explorer panel option, and is five by default.
- **Saved Results** – This stores any result set you have moved or copied to it, or saved by using **Save** from the context menu. Results are not removed from the store until you do it yourself.
- **Merged Results** – The store of result sets obtained by *merging* (see below).
- Optional **Folders** – The **New Folder** item on the context menu lets you add as many main branches as you wish. All behave like **Saved Results**. In other words, Saved Results is the default folder, and the folders you add and name yourself let you put more organization into your store of saved results. Result sets are added by dragging.

Besides dragging, copying and deleting, you can perform the following operations with result sets:

- **Comparison**. The point of getting profile results is usually to improve your code. So, between different builds, it can be extremely helpful to have a simple way to compare result sets on the values of interest. This is what the Explorer panel's Compare operation allows you to do. See *Comparing and Merging Results*.
- **Merging**. As long as you are profiling the same build of your application, or builds where the code of interest has not changed, combining result sets can be a major help in getting better statistics. This is called *merging*. Also, since you can merge your results later, you are free to do shorter and simpler profile runs on separate aspects of your application. For detailed instructions on how to merge two or more result sets, see *Comparing and Merging Results*.
- **Export and import**. Result sets can be exported to a file, which allows transferring specific result sets from one machine to another. For more compactness in the latter use, the export format is binary. Exporting any single result set is available from the context menu as **Save to File** and importing as **Load From File**. See *Exporting Results*.

Explorer Panel Options

The **Explorer** panel offers an extensive complement of customizable options. You can check or change these options using the **Explorer Options** dialog. To call it, do any of the following:

- *AQtime standalone:*
  - Select **Options** from the context menu of the Explorer panel.
  - Select **Options | Options** from AQtime's main menu (this will call the Options dialog) and then choose **Analysis Results | Explorer** from the tree view on the left of the dialog.

- *AQtime integrated into Visual Studio:*
  - Select **Options** from the context menu of the Explorer panel.
  - Right-click **Explorer** in Visual Studio's **Solution Explorer** and select **Properties** from the context menu.
  - Select **Tools | Options** from the main menu of Visual Studio and then choose **AQtime | Analysis Results | Explorer** group in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Options** from the context menu of the Explorer panel.
  - Select **Profile | Options** from the main menu of Borland Developer Studio (this will call the Options dialog) and then choose **Analysis Results | Explorer** from the tree view on the left of the dialog.

The following options are available:

- **Number of recent results to keep** – Sets the number of entries in the Last Results list. The default value is 5.
- **Always set up Compare parameters** – Sets whether to show the **Compare Settings** dialog every time you compare results. Enabled by default.
- **Show results for all profilers** – When this is disabled (the default state), the Explorer panel only includes results for the currently selected profiler. When this is enabled, each profiler is shown as a main branch in the panel, with its results in tree view under that branch.
- **Auto-merge** – Auto-merge is a feature where a special, separate result set accumulates profiling results by merging each new result set with the previous one. Suppose that the Explorer panel holds no results. After you profile your application and obtain profiler results, the auto-merge result will be the same as the first profiling result. However, after you profiled your application for the second time, the auto-merge result set will be the merged result of the first and the second results. After the third profiler run, AQtime will merge the new result set with the two preceding ones and show it as the auto-merge result, etc.
  - **Active** – Enables or disables the auto-merge feature. Disabled by default.
  - **Folder name** – Specifies the name for the special auto-merge result, when the auto-merging is enabled. The auto-merge result is shown in the **Merged Results** folder in the Explorer panel.

Monitor Panel

Monitor Panel - Description

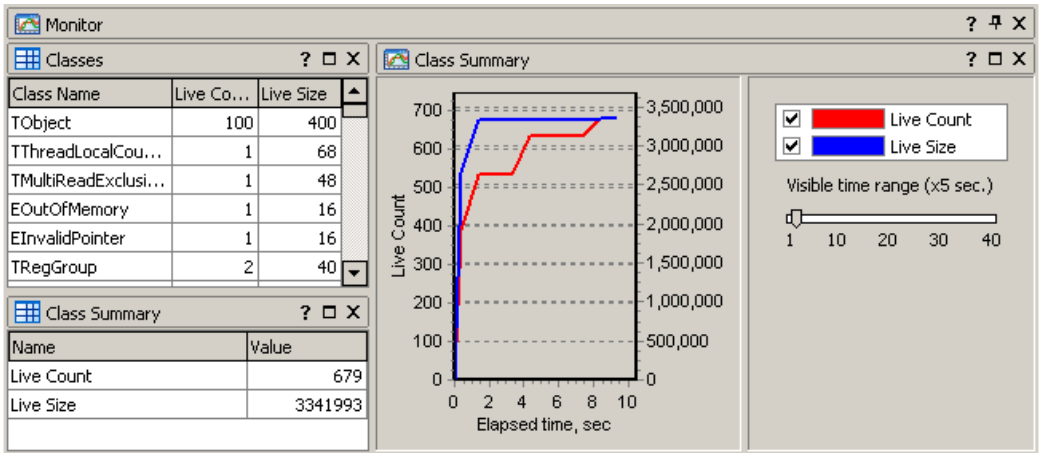
The **Monitor** panel is aimed at monitoring real-time memory usage. It presents the output of the Allocation profiler using grids, graphs and histograms in real time. To display the Monitor panel, do any of the following:

- *AQtime standalone:*
  - Select **Monitor** from the **View | Other Panels** menu.
  - Select **Monitor** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.
  - Select **Monitor** from the **Assistant** panel.
- *AQtime integrated into Visual Studio:*

- Select **Monitor** from the **Select Panel** dialog, which is called by selecting **Profile | Panel List**.
- Select **Monitor** from Visual Studio's **Solution Explorer**.
- Select **Monitor** from the **Assistant** panel.

- *AQtime integrated into Borland Developer Studio:*
  - Select **Monitor** from the **View | Profile Windows | Other** menu.
  - Select **Monitor** from the **Assistant** panel.

Here is an example of the Monitor report:



To enable or disable monitoring during the run, you can use the **Active** item of the context menu. The *Active* option of the panel is an alternative way to turn on/off the monitoring. If you disable (pause) monitoring and then activate (resume) it, the collected data will not be lost. Instead, new data data will be added to the existing data. The information that was generated during the pause will not be included in the report. The *Update interval* option specifies the time interval, in which the Monitor updates the displayed information.

By default, the Monitor panel displays several predefined panes. You can hide panes that you do not need at the moment and display them again using the **Show Panes** submenu of the context menu. You can also dock and undock these panes within the Monitor panel in the same manner as you dock and undock other AQtime panels (see *Docking*). To enable or disable docking, use the **Allow Docking in Monitor** item of the Monitor's context menu. You can also create your own panes with graphs and histograms on them (see below for more information on how to do this).

Each pane can display result data using any of the following views:

View	Description
Counter	In the **Counter** view, all the data is displayed as grids. You can move grid columns and change their width like you do this in other AQtime panels (see *Arranging Columns, Lines and Panels*). Grids' records are called ***series***. In comparison with other views that graphically illustrate the results and give their common image, the Counter view is suitable for precise analysis of the results.
Histogram	The **Histogram** view displays series within a chart and visually demonstrates the difference between series values. Multiple display styles are available when using this view. You can choose an appropriate style by selecting **Chart Style** from the context menu: **Bar**, **Area**, **Pie** or **Point**. Using the **Show Marks** context menu item, you can display or hide marks in the given histogram. If you want to apply the 3D style to the histogram, enable **3D View** in the context menu.
Graph	The **Graph** view displays series within a graph during the application functioning.

	This allows you to compare certain series by their values at definite moments of time. Sometimes, values of series on the same graph are too different, which lets you easily see the dynamic of one series' values, but turns the graph of another series to a straight line shown in parallel with the X axis (for instance, if values of one series are close to 10,000, while values of another series are about 100). To make both graphs more visual, enable **Separate Y Axis** on the context menu. This will set an individual Y axis for each series whose values are displayed on the graph and fit the height of each series' graph within the height of the visible area. Thus, the values 9,500 and 95 will be displayed at the same vertical level. If Separate Y Axis is turned on, you will see two vertical scales for the first two series displayed: one scale is on the left (as usual) and another is on the right. The other series will be displayed without scales and you will only have to guess what value this or that series actually has. In short, using this mode makes sense if only two series are shown on the graph. To mark a specific point in time in the graph, just click the needed point. If the panel's *Show event marks* option is on, this will display a vertical line at the respective X coordinate. This line is called an *event mark*. The style of this line (solid, dot, dash-dot, etc.) is determined by the *Event mark style* option of the Monitor panel. If you place the mouse coursor over that line, you will see the results that correspond to the chosen time for all the series displayed in the graph. You can create as many event marks as you want. Event marks are also created automatically each time you pause/resume monitoring (via the *Active* option of the Monitor panel), pause/resume profiling (via the **Pause** and **Resume** buttons) or stop profiling (via the **Terminate** button). These event marks give you information on when monitoring or profiling was paused or stopped and when it was resumed.

The Monitor panel includes a flexible system of settings which allows you to make the panel display only the values you need. This speeds up the monitoring process and helps you avoid analyzing large amounts of information which does not serve your specific testing requirements. The panes that use the graphical views (Graph and Histogram) display resultant data for a number of series comparing the same sole value in them. The panes that use the Counter view display results for a number of series and calculate several values for each series.

The contents of the posted reports depends on the profiler used and the application being profiled. Currently, the Monitor works with the **Allocation** profiler only (see *Using the Monitor Panel With the Allocation Profiler*). When the Allocation profiler is running, the Monitor panel tracks creations and deletions of class instances as well as allocations and deallocations of memory blocks in the profiled application. The profiler provides two lists of series to be reported in the Monitor panel: the list of classes and the list of the summary values for these classes. The Monitor panel displays these series on the **Classes** and **Class Summary** panes. These panes are predefined; you can neither delete them, nor create their counterparts. The profiler determines a constant list of values to be calculated for each series in the given list. For example, for every class displayed on the Classes pane for the Allocation profiler, the Monitor displays two values - **Live Count** and **Live Size** - calculated during profiling.

The series list is empty before the application's first run and it is populated during profiling. After the application starts running, the Monitor panel saves all the information on series that were used in the current run. During and after the profiler run, you can specify which series to display in the given Graph or Histogram view. To do this, choose **Select Series** from the context menu, and in the resulting **Select Series** dialog enable the needed series and disable those you wish to hide. Alternatively, if the **Show Legend** item is turned on in the context menu, you can choose which series to display via the legend box that is shown to the right of the chart. Like in the Select Series dialog, you can enable or disable all the series that are listed in the legend box at once. To do this, right-click within the legend box and choose **Select All** or **Unselect All** from the context menu. To invert the selection of the series in the legend box, choose **Invert Selection** from the context menu. The splitter

that separates the legend box and the chart is movable. The Select Series dialog and the legend box also let you choose a specific color for each series. The series settings you made using the Select Series dialog or the legend box are preserved in further profiler runs.

Unlike the Graph and Histogram views, the Counter view always displays all the series about which the profiler "knows".

For each value column that is shown in the Counter view, you can create a custom pane with the Graph or Histogram view. To create a new pane:

- Select **New Chart Pane** from the context menu.
- Use the subsequent **New Chart Pane** dialog to specify the chart's type (a graph or a histogram), the chart's data source (the series list and the value column to use), the columns whose values will be used to identify series in the chart, etc.

Note that once you have created a new histogram pane, the chart that is displayed in it will include all the available series except for those whose value is zero (they will be disabled in the legend box and in the **Select Series** dialog automatically). If you explicitly enable these exceptional series, their zero-value histograms will appear in the chart.

To delete the selected pane from the Monitor panel (it is possible to do for custom panes with the Graph or Histogram view), choose **Delete Chart Pane** from the context menu. Upon doing this, the deleted pane will no longer be available in the Show Panes list.

To zoom in the contents of a pane that uses the Graph or Histogram view, drag the mouse cursor from the upper left corner of an imaginary rectangle to its lower right corner. To zoom out the view, drag the cursor in any other diagonal direction of that rectangle. To cancel zooming and display the selected chart in its normal size, select **Zoom to 100%** from the context menu.

To zoom in or out the contents of a graph's horizontal axis, drag the slider of the magnifier, which is visible at the lower right corner of the given pane if the *Display magnifier* option is enabled.

Monitor Panel Options

The **Monitor** panel offers an extensive complement of customizable options. You can check or change these options using the **Monitor Options** dialog. To call it, do any of the following:

- *AQtime standalone:*
  - Select **Options** from the context menu of the Monitor panel.
  - Select **Options | Options** from AQtime's main menu and then choose the **Profile Time | Monitor** group in the ensuing Options dialog.
- *AQtime integrated into Visual Studio:*
  - Select **Options** from the context menu of the Monitor panel.
  - Right-click **Monitor** in Visual Studio's **Solution Explorer** and select **Properties** from the context menu.
  - Select **Tools | Options** from the main menu of Visual Studio and then choose the **AQtime | Profile Time | Monitor** group in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Options** from the context menu of the Monitor panel.
  - Select **Profile | Options** from the main menu of Borland Developer Studio and then choose the **Profile Time | Monitor** group in the ensuing Options dialog.

The following options are available:

- **Event Marks**
  - **Show event marks** - Specifies whether to display event marks (i.e. vertical lines used to mark definite time points in graphs).
  - **Event mark style** - Sets the style of vertical lines that designate event marks in graphs. Available values: *Solid*, *Dot*, *Dash*, *Dash-Dot* and *Dash-Dot-Dot*.
- **Active** - Sets whether the monitoring is enabled in the panel. This option is also controlled by the **Active** item of the panel's context menu. To learn how changing this option affects displayed results during the run, see *Monitor Panel*.
- **Update interval** - Sets the time interval (in seconds) after which AQtime updates the contents of the Monitor panel.
- **Visible time range** - Sets the length (in seconds) of the horizontal axis range that is visible in graphs.
- **Display magnifier** - Sets whether to display the horizontal axis magnifier that is used in graphs.

Using the Monitor Panel With the Allocation Profiler

The **Allocation** profiler traces creation of objects and allocation of memory blocks during the application run. When the **Monitor** panel and the Allocation profiler are used together, the panel shows information on existing class instances and memory blocks in real time. This data is displayed in two data grids (Classes and Class Summary). Each of them is located on a separate pane.

The **Classes** pane holds a list of the application's classes and calls to C++ and VCL memory-allocating routines per profiled module. This information is given in two columns: **Class Name** and **Module Name**. That is, each record (series) in this list corresponds to a class whose instances were created in the given module during the application execution. For memory blocks that were allocated via C++, VCL or VB memory management routines and statements called from the given module, the Class Name column holds the *C++ native memory*, *VCL native memory* or the *VB native memory* value correspondingly. For each class, two values are calculated: **Live Count** (the current number of instances of the given class or the current number of memory blocks allocated by calls to memory management routines of the respective category, C++ or VCL) and **Live Size** (the total amount of memory in bytes that is currently occupied by all these instances or memory blocks).

Note: Depending on the Allocation profiler's *File names with path* option, which is displayed as a toolbar button at the top of the **Report** panel, the **Module Name** column of the **Monitor** panel shows names of modules with or without paths to them.

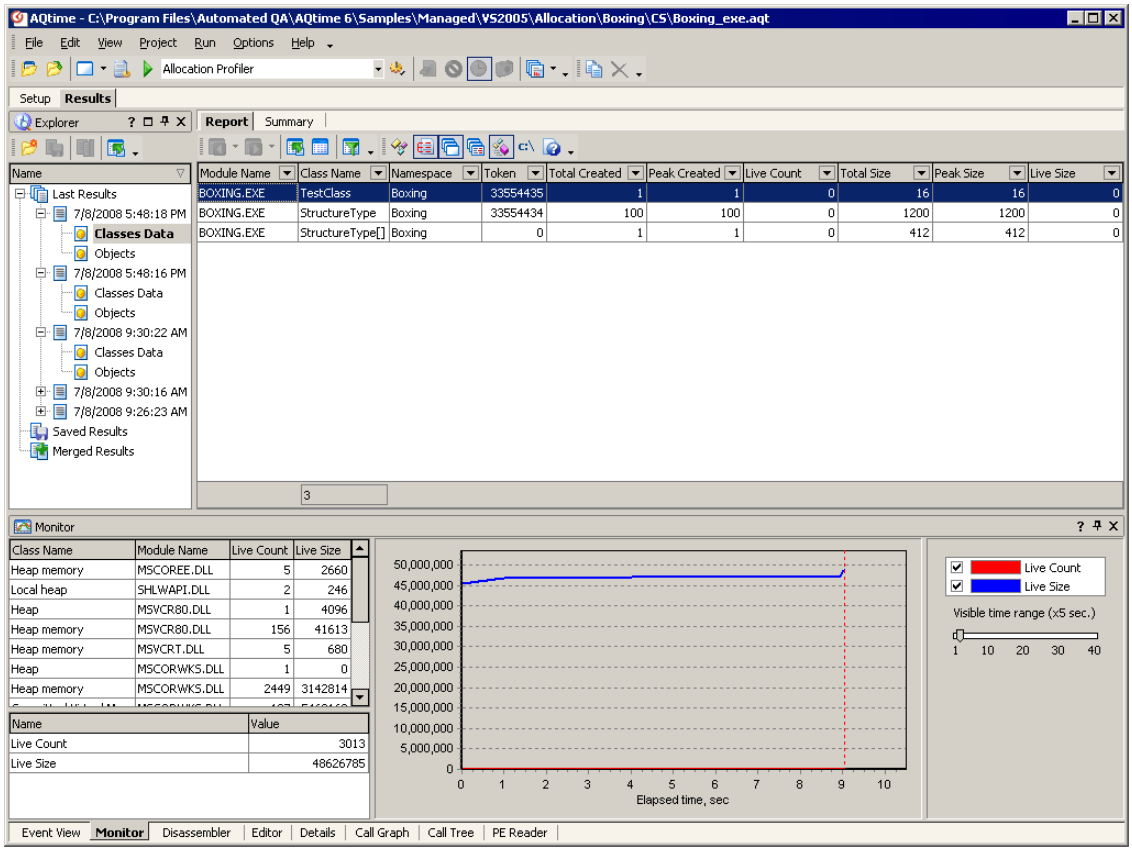
Another option of the Allocation profiler, *Show all loaded classes* (it is also displayed on the Report panel's toolbar), specifies whether the series list in the Monitor panel includes all the classes being profiled or only the classes whose instances were created by the current moment.

The **Class Summary** pane displays a list of summary values for all the classes and calls to memory management routines that were profiled. This pane displays the following parameters: **Live Count** (the current number of instances of each profiled class summed up for all the classes engaged in profiling plus the current number of memory blocks allocated by a call to a memory management routine summed up for all the routines engaged in profiling) and **Live Size** (the total amount of memory in bytes that is currently occupied by all these instances and memory blocks). The value of each parameter is shown in the **Value** column. This information lets you quickly determine the total number of class instances and memory blocks that exist at the moment and how much memory they occupy.

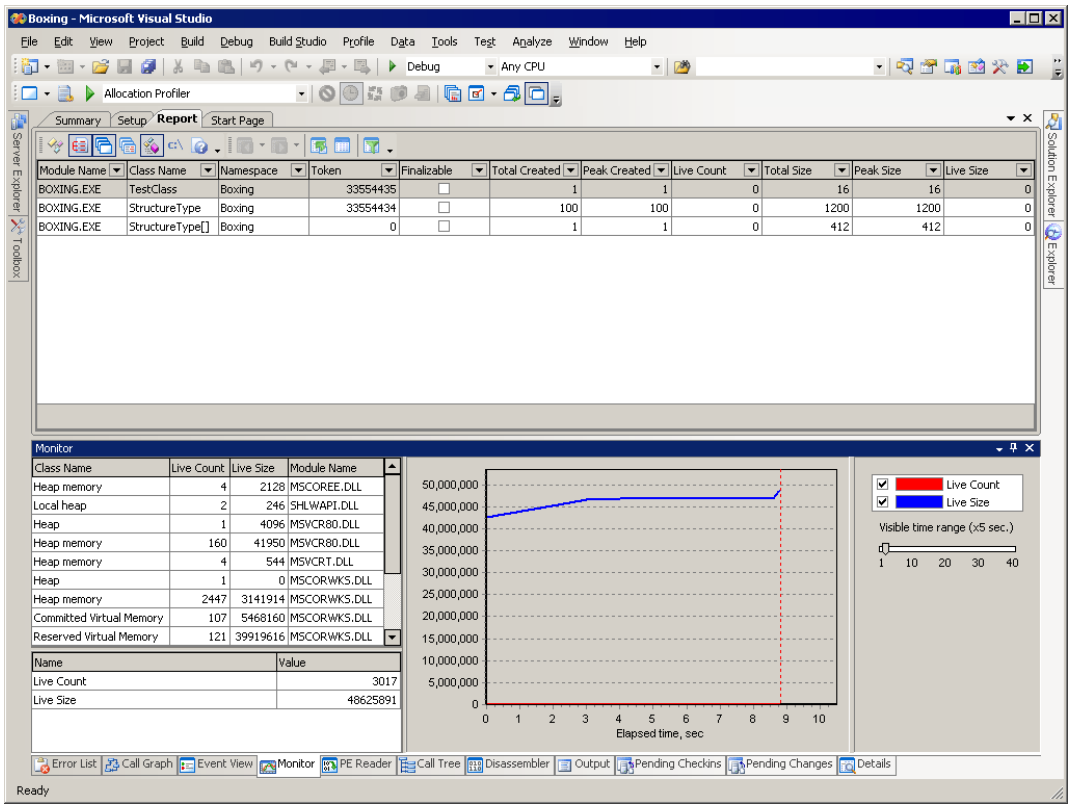
Note: The amount of used memory displayed by AQtime may differ from the amount of memory shown

for your application in the Task Manager. The reason is that AQtime displays the memory that is currently allocated by the application's memory manager for *all live objects being profiled* (the Allocation profiler traces only those objects, whose classes are included in profiling areas). In the Task Manager window, you see the memory size that is allocated by the operating system's memory manager for the application. Some part of this memory may not be used at the moment, but still it is allocated by the application's memory manager (for instance, for future use). In certain cases, deallocated memory blocks may not be returned to the operating system's memory manager, so the operating system "thinks" that these blocks are still being allocated by the application. There are also other possible reasons. So, the difference you see is caused by the peculiarities of memory management in the operating system and in the application.

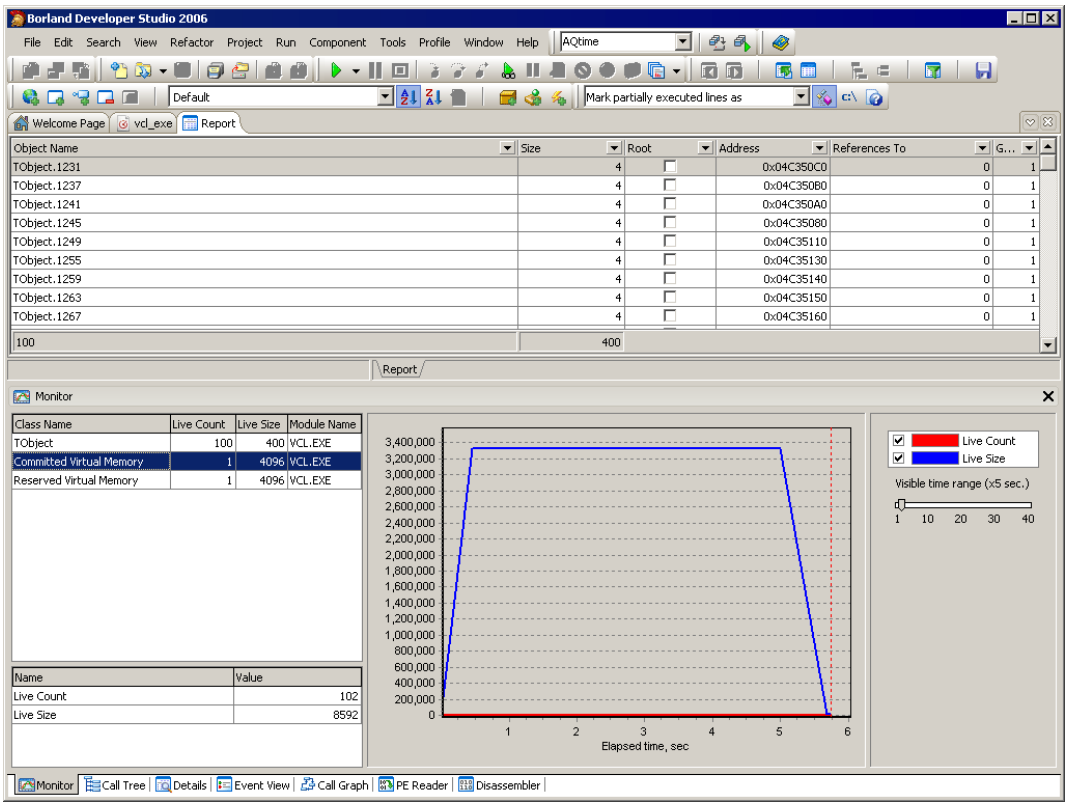
You can also use a graph or a histogram to display the above-mentioned values in the Monitor panel. To do this, you need to create custom chart panes in the panel as it is described in the *Monitor Panel* topic.


AQtime - C:\Program Files\Automated QA\AQtime 6\Samples\Managed\VS2005\Allocation\Boxing\CS\Boxing_exe.aqt
File Edit View Project Run Options Help
Allocation Profiler
Setup Results
Explorer
Report Summary
Name
Last Results
7/8/2008 5:48:18 PM
Classes Data
Objects
7/8/2008 5:48:16 PM
Classes Data
Objects
7/8/2008 9:30:22 AM
Classes Data
Objects
7/8/2008 9:30:16 AM
7/8/2008 9:26:23 AM
Saved Results
Merged Results
Module Name | Class Name | Namespace | Token | Total Created | Peak Created | Live Count | Total Size | Peak Size | Live Size
BOXING.EXE | TestClass | Boxing | 33554435 | 1 | 1 | 0 | 16 | 16 | 0
BOXING.EXE | StructureType | Boxing | 33554434 | 100 | 100 | 0 | 1200 | 1200 | 0
BOXING.EXE | StructureType[] | Boxing | 0 | 1 | 1 | 0 | 412 | 412 | 0
Monitor
Class Name | Module Name | Live Count | Live Size
Heap memory | MSCOREE.DLL | 5 | 2660
Local heap | SHLWAPI.DLL | 2 | 246
Heap | MSVCR80.DLL | 1 | 4096
Heap memory | MSVCR80.DLL | 156 | 41613
Heap memory | MSVCRT.DLL | 5 | 680
Heap | MSCORWKS.DLL | 1 | 0
Heap memory | MSCORWKS.DLL | 2449 | 3142814
Name | Value
Live Count | 3013
Live Size | 48626785
50,000,000
45,000,000
40,000,000
35,000,000
30,000,000
25,000,000
20,000,000
15,000,000
10,000,000
5,000,000
0
0 1 2 3 4 5 6 7 8 9 10
Elapsed time, sec
Live Count
Live Size
Visible time range (x5 sec.)
1 10 20 30 40
Event View Monitor Disassembler Editor Details Call Graph Call Tree PE Reader

AQtime standalone


Boxing - Microsoft Visual Studio
File Edit View Project Build Debug Build Studio Profile Data Tools Test Analyze Window Help
Debug
Any CPU
Allocation Profiler
Summary Setup Report Start Page
Module Name Class Name Namespace Token Finalizable Total Created Peak Created Live Count Total Size Peak Size Live Size
BOXING.EXE TestClass Boxing 33554435 1 1 0 16 16 0
BOXING.EXE StructureType Boxing 33554434 100 100 0 1200 1200 0
BOXING.EXE StructureType[] Boxing 0 1 1 0 412 412 0
Monitor
Class Name Live Count Live Size Module Name
Heap memory 4 2128 MSCOREE.DLL
Local heap 2 246 SHLWAPI.DLL
Heap 1 4096 MSVCR80.DLL
Heap memory 160 41950 MSVCR80.DLL
Heap memory 4 544 MSVCRT.DLL
Heap 1 0 MSCORWKS.DLL
Heap memory 2447 3141914 MSCORWKS.DLL
Committed Virtual Memory 107 5468160 MSCORWKS.DLL
Reserved Virtual Memory 121 39919616 MSCORWKS.DLL
Name Value
Live Count 3017
Live Size 48625891
Live Count
Live Size
Visible time range (x5 sec.)
Elapsed time, sec
Error List Call Graph Event View Monitor PE Reader Call Tree Disassembler Output Pending Checkins Pending Changes Details
Ready

AQtime integrated into Visual Studio


Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile Window Help
AQtime
Default
Mark partially executed lines as
Welcome Page vcl_exe Report
Object Name Size Root Address References To
TObject.1231 4 0x04C350C0 0 1
TObject.1237 4 0x04C350B0 0 1
TObject.1241 4 0x04C350A0 0 1
TObject.1245 4 0x04C35080 0 1
TObject.1249 4 0x04C35110 0 1
TObject.1255 4 0x04C35130 0 1
TObject.1259 4 0x04C35140 0 1
TObject.1263 4 0x04C35150 0 1
TObject.1267 4 0x04C35160 0 1
100 400
Report
Monitor
Class Name Live Count Live Size Module Name
TObject 100 400 VCL.EXE
Committed Virtual Memory 1 4096 VCL.EXE
Reserved Virtual Memory 1 4096 VCL.EXE
Name Value
Live Count 102
Live Size 8592
Live Count
Live Size
Visible time range (x5 sec.)
Elapsed time, sec
Monitor Call Tree Details Event View Call Graph PE Reader Disassembler

AQtime integrated into Borland Developer Studio

PE Reader Panel

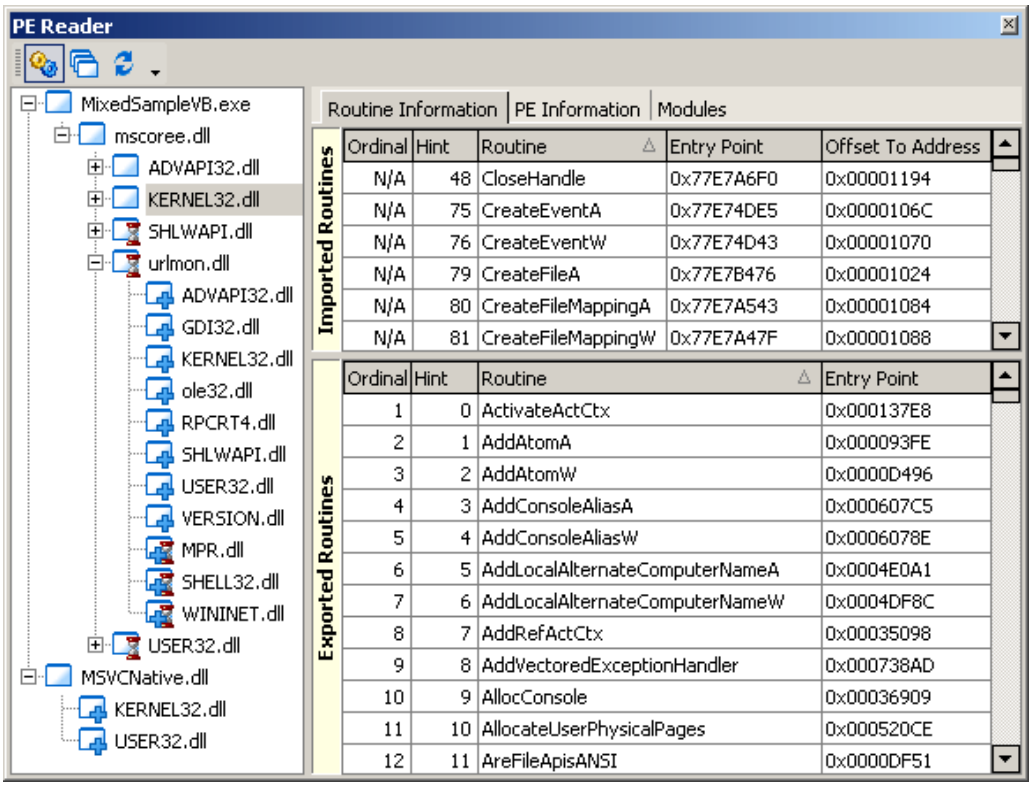
The **PE Reader** panel serves for the analysis of module relationships in the profiled application. When loading a project, PE Reader analyzes the modules linked to the application at load-time, builds a tree-like structure of these modules and displays detailed information about each module.

PE Reader works both with Windows and .NET executables. It does not require the application be compiled with debug information. It simply analyzes the application code and helps you --

- Determine what modules are required for the running application.
- Determine defective files.
- Determine what functions each module imports and exports.
- Examine detailed information about the functions that are used by the application: entry points, function addresses in a module, etc.
- Examine detailed information about the modules that are used by the application: operating system version, module version, image file type, debug info existence, entry point, image base address, processor type, etc.
- Determine whether a function belongs to a module, etc.

To display the PE Reader panel, do any of the following:

- *AQtime standalone:*
  - Select **PE Reader** from the **View | Other Panels** menu.
  - Select **PE Reader** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.
  - Select **PE Reader** from the **Assistant** panel.
- *AQtime integrated into Visual Studio:*
  - Select **PE Reader** from the **Select Panel** dialog, which is called by selecting **Profile | Panel List**.
  - Select **PE Reader** from Visual Studio's **Solution Explorer**.
  - Select **PE Reader** from the **Assistant** panel.
- *AQtime integrated into Borland Developer Studio:*
  - Select **PE Reader** from the **View | Profile Windows | Other** menu.
  - Select **PE Reader** from the **Assistant** panel.



PE Reader scans all application modules recursively beginning with the main module. If a module, say a dynamic link library, imports some functions from another dynamic link library, PE Reader analyzes the latter and displays it as a child node of the “parent” DLL in the module tree. The recursion continues until all the used modules are processed. If, during the recursion, PE Reader meets a module that has already been reviewed, it does not analyze this module. PE Reader marks it with a special icon ( ) and displays the module without “children” in the module tree, that is, the icon means that the module has been analyzed somewhere before. To view “child” modules of such a module, select this module in the module tree and then choose **Show Imported Modules** from PE Reader’s context menu or from the PE Reader toolbar.

To add a module displayed in the module tree to the current AQtime project, right-click that module and choose **Add Module to Project** from the context menu.

There can be several versions of the module that can be loaded by the parent executable. The version to be loaded is specified by the manifest of that parent executable. Since AQtime does not “know” which module will be loaded, it displays the version that best matches the specified version.

Below is a description of possible module’s icons:

Icon	Description
	Ordinary module.
	Duplicated module
	Delay-loaded module.
	Defective or unavailable module. PE Reader uses this icon if the module cannot be executed or if it is absent.

Note: A combination of marks is possible. For instance, the icon means the module is duplicated and delay-loaded.

To update the module tree, select **Reload Modules Tree** from the PE Reader toolbar or from the context menu. The refresh is necessary, for example, if initially an imported module was absent, but then it was created (that is, it became available).

Information about each module is shown on the following tabbed pages:

- **Routine Information** - Holds tables of imported and exported functions.
- **PE Information** - Displays information about headers and sections of the module selected in the module tree.
- **Modules** - Displays a list of modules that are linked to the profiled executable at load-time.

The rest of this topic provides detailed information on the tabbed pages.

Routine Information Page

The Routine Information page of the PE Reader panel displays two lists of functions: the lower list, **Exported Routines**, holds functions exported by the module currently selected in the module tree; the upper list, **Imported Routines**, holds functions that are called by the "parent" module from the selected module.

Columns	Description
Entry Point	Routine's address in the module. For imported routines, this address includes the module's base address that is specified in the module's header. For exported routines, this address does not include the module's base address.
Hint	Hint value of a routine. This value is a function index in the array of functions exported by a module. The system uses this value for rapid search of a function in the module.
Ordinal	Holds the routine's ordinal number. The Ordinal column of the Imported Routines table may hold the "N/A" value that means the routine is imported by name.
Offset to Address	This column is in the Imported Routines list only. It holds the offset of the routine's address in the import address table of the "parent" module. The import address table (IAT) is used to call a routine kept in another module. When an executable (EXE or DLL) calls a routine stored in another module, control does not go directly to the desired routine. Instead, it goes to an instruction like `JMP DWORD PTR [`*`XXXXX`*`]` (you can check this by tracing your application execution in a debugger). The `JMP` instruction transfers control to the address of the desired routine. The *XXXXX* value specified in brackets is a sum of IAT's address and an offset, at which the entry point of the desired routine is stored in the IAT. The Offset to Address column holds that offset.
Routine	Name of the routine. The Routine name of the Imported Routines table may hold the "N/A" value that means that the function is imported by ordinal number. The routine name may be decorated or undecorated according to the state of the **Undecorate routine names** toolbar item.

Both Exported Routines and Imported Routines tables can be arranged at your desire: you can change the column sizes, remove columns from or add them to the tables, sort records on any column, etc. (See *Arranging Columns, Lines and Panels* in on-line help). You can also search for information in the tables using the incremental search mechanism (see *Searching Results*).

To view the source code of a routine shown in the Exported or Imported Routines list, double-click this routine and then switch to the Editor panel. Note that you will be able to view the routine's source code only if the module holding the routine matches the following requirements:

- It must be included in the current AQtime project (if the module does not belong to the project, AQtime will ask you to add it there).
- It must be compiled with debug information (see *How AQtime Profiles Use Metadata and Debug Information*).

To view the binary code of a routine, double-click this routine in the Exported Routine or Imported Routines list and switch to the Disassembler panel. Note that you will be able to view the binary code only if the module holding the routine is included in the current AQtime project. If the module was compiled without debug information, the panel may show more assembler instructions than the routine's binary code actually contains. This happens because without debug info it is impossible to determine the exact size of the routine's binary code, so AQtime has to resort to its own algorithm, which is less accurate than debug info (the algorithm may report that the routine contains more binary instructions than it actually has).

PE Information Page

This page displays detailed information about the headers and sections of the module that is currently selected in the modules tree.

Headers

The header of a module consists of several parts (or several headers). They hold detailed information about the module. All the modules have the following headers:

Header	Description
DOS Header	The header that existed in all DOS executable applications plus the field that indicates the offset of PE Header.
PE Header	Holds information about the processor type, the number of application sections, and the date of file creation and file attributes.
Optional Header	Holds specific information used by the operating system, for example, the version number of the required operating system, control sum, image base address, etc.

For more information about the structure and contents of PE Headers, see MSDN Library (on-line version is available at http://www.msdn.microsoft.com).

Sections

Sections are "segments" of code or data within an executable. In general, a file can include any section with an arbitrary name and purpose, but some sections, for example, *debug* or *rsrc*, have specific meaning. For detailed information, see MSDN Library. Note that Windows NT limits the number of sections to 96.

For each section, PE Reader displays the following information:

- Virtual address of a section in the process address space.
- Relative size of the section body.
- The offset of the section body in a file.
- Section attributes.

Referenced Assemblies

For .NET modules, the PE Information panel holds a list of assemblies the selected module refers to. For each referred assembly, PE Reader displays its module name, version, the Public Key value and the culture attribute. Use this information to determine what version of an assembly your executable requires.

Modules Page

The Modules page displays the modules that are linked to the profiled executable at load-time (that is, the module's functions are encapsulated by the executable). Use this page to quickly view a list of modules necessary for application execution. For each module, the following information is available:

Columns	Description
Module	Name of the module.
File Size	Size of the module in bytes.
File Version and **Product Version**	These fields are added by your compiler. They specify the full version numbers of the module and of the entire application (product). These versions include the build number.
Image Version	The version of the executable file.
Link Time Stamp	Date and time of file creation. Do not confuse them with the date and time file attributes. Link Time Stamp is the date and time of file creation that are specified in the executable header. These values are written there by the linker.
Machine	The machine (CPU) type. The module can be run on the specified machine only or on a system that emulates it.
OS Version	The version of the required operating system.
Path	Path to the module.
Preferred Address	The preferred address for loading the module in memory.
Subsystem	The subsystem, which is required to run the module: Windows GUI, console subsystem, Posix character subsystem, device driver, etc.
Subsystem Version	The version of the subsystem is required to run the module.

The Modules table can be arranged at your desire: you can change the column sizes, sort records on any column, etc. For more information on this, see *Arranging Columns, Lines and Panels* in on-line help. You can also search for information in the table using the incremental search feature (see *Searching Results*).

Report Panel

Report Panel - Description

The **Report** panel is the basic display for profiling results. Specific elements can be selected in Report to define what will be displayed in turn in other panels: **Editor**, **Call Graph**, **Call Tree** and **Details** (see *AQtime Panels*). To display the Report panel, do any of the following:

- *AQtime standalone:*
  - Select **Report** from the **View** menu.
  - Select **Report** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.

- Select Report from the **Assistant** panel.

- *AQtime integrated into Visual Studio:*
  - Select **Report** from the **Select Panel** dialog, which is called by selecting **Profile | Panel List**.
  - Select **Report** from Visual Studio's **Solution Explorer**.
  - Select **Report** from the **Assistant** panel.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Report** from the **View | Profile Windows** menu.
  - Select **Report** from the **Assistant** panel.

The contents of the Report panel depend on the profiler used to generate the results on display. These are normally those of the last run, but they can also be retrieved from previous runs through the **Explorer** panel. To get help on the profiler that generated these results, press Ctrl-F1 or choose **Help On Selected Profiler** from the Report context menu.

Depending on the current profiler and the profile selected in the Explorer panel, each row in the Report panel gives results for one profiled routine, object, class, etc. As you shift from one line to another and check the ensuing details in other panels, your movements are tracked, so that you can retrace your steps back and forth using the **Display Previous** and **Display Next** buttons on the **Report** toolbar. When you press the down arrow button to the right of these buttons, you can see a list of steps you can return to. To better identify the desired step, you can configure the format that will be used for items in this list. To do this, use the **Customize Navigation** dialog, which is called by pressing **Customize Navigation** from the context menu.

For each numerical column, the footer (the last grid row) can hold a summary field for all values in the column. The summary field can display the sum, average, maximum, minimum or count for the items displayed above it, or the summary can be disabled for the given column. To change the format for the selected column's footer, right click on it and choose the desired format from the context menu. Note that the footer is hidden if the *Show footer* option is off. If records in the Report panel are grouped by one or several columns, such summary fields can be also displayed for each group node. This is possible if the *Show group footer* is enabled.

By default, AQtime does not display all available columns in the Report panel (if it did the panel will be overloaded with results). You can add columns to and remove them from the panel at any time. For more information on this, see *Adding and Removing Columns*. Note that you can easily arrange columns (move them, change their size, etc.) the same way you can do this in other AQtime panels. See *Arranging, Columns, Lines and Panels*. Besides supporting these standard adjustments, the Report panel lets you:

- Change a column’s width so that the column becomes wide enough to display its contents and caption. To do this, select **Adjust Column Width** from the context menu.
- Change the font color for a column, its text alignment, its format string, etc., by selecting **Format Columns** from the context menu, which will call the **Format Columns** dialog.
- Alternatively, change text alignment in a column by selecting **Alignment** from the column header's context menu.
- Change data display format for a column (**Value**, **Percent**, **Graph Bar** or **Color**) by selecting **Display Format** from the column header's context menu. See *Displaying Results in the Report and Details Panels.*

The mentioned features specify how results are displayed. The following features serve for the result analysis:

- Sort results by one or several columns. See *Sorting Results*.

© 2010 AutomatedQA Corp.

- Find records (lines shown) by some key value, by using the Find dialog or the *Incremental Search* mechanism. See *Searching Results*.
- Filter results on-the-fly and create complex filter conditions. See *Filtering Results*.
- Apply pre-defined result views to instantly get a combination of filter and display settings. See *Result Views*.
- Group results into a subtree when they share one or several common key values by one or more Report columns. The **Format Columns** dialog will let you define how values are calculated for display in the group's top (summary) line. The usual sort-on-column feature will work on the summary line to sort entire groups. Note that each grouping column has its own summary settings. See *Grouping Results*.

Note that the **Report** panel footer shows summary values for some panel columns. By default, the summary values are calculated against all panel rows. If you select two or more rows in the panel, AQtime will calculate the summary for the selected rows only.

Report Panel Options

The **Report** panel offers several customizable options. You can check or change these options using the **Report Options** dialog. To call it, do any of the following:

- *AQtime standalone:*
  - Select **Options** from the context menu of the Report panel.
  - Select **Options | Options** from AQtime's main menu and then choose the **Analysis Results | Report** group in the ensuing Options dialog.
- *AQtime integrated into Visual Studio:*
  - Select **Options** from the context menu of the Report panel.
  - Right-click **Report** in Visual Studio's **Solution Explorer** and select **Properties** from the context menu.
  - Select **Tools | Options** from the main menu of Visual Studio and then choose the **AQtime | Analysis Results | Report** group in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Options** from the context menu of the Report panel.
  - Select **Profile | Options** from the main menu of Borland Developer Studio and then choose the **Analysis Results | Report** group in the ensuing Options dialog.

The following options are available:

- **Activate on generating results** – Specifies whether AQtime will switch to the Report panel after the results have been generated or loaded. Enabled by default.
- **Show footer** – Sets whether a footer row will be added to show column totals.
- **Show group footer** – In the **Format Columns** dialog you can specify how AQtime calculates the summary values for a column when results are not grouped by this column. If the *Show group footer* option is enabled these summary values are displayed in each group footer. Else, the group footer area is hidden.

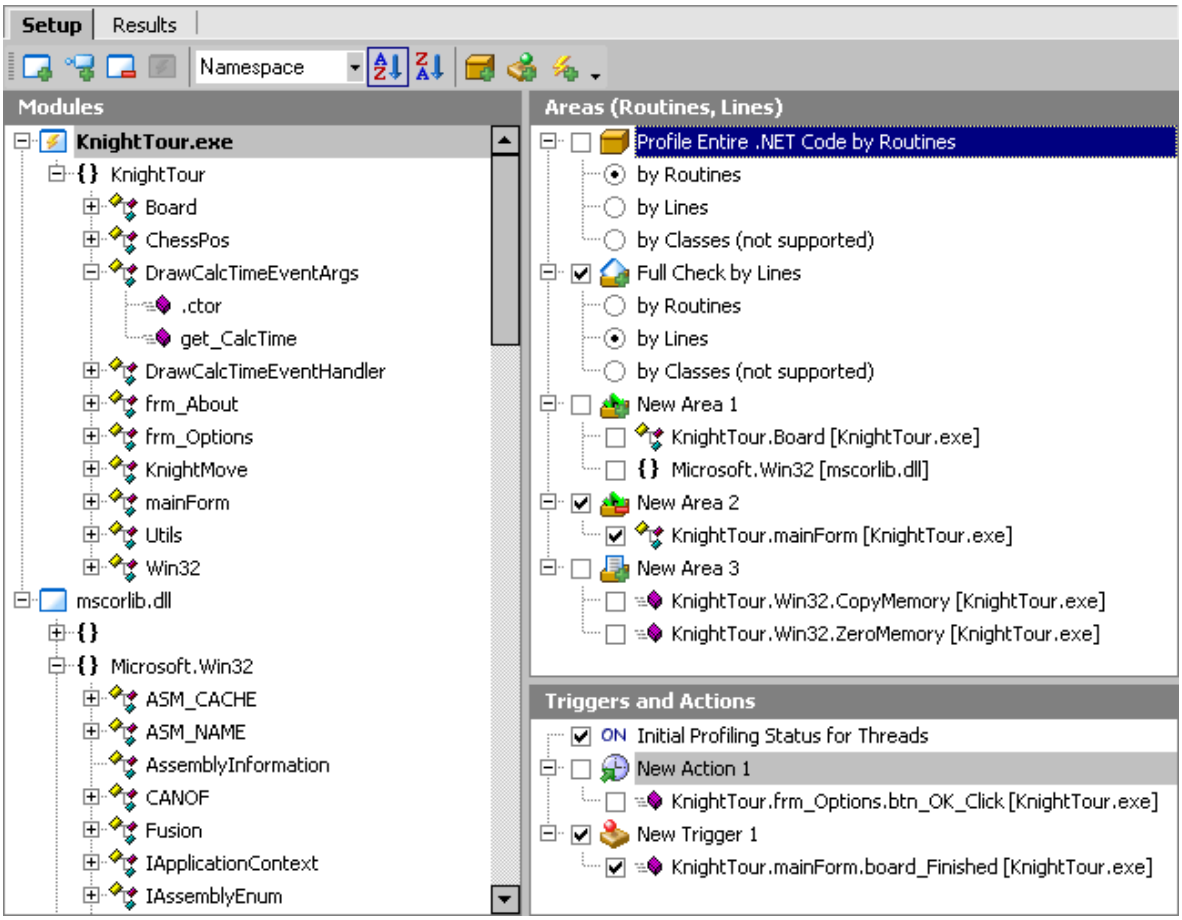
Setup Panel

Setup Panel - Description

The **Setup** panel is your tool for defining what portions of code to profile, and when. See *Controlling What to Profile*. To display the Setup panel, do any of the following:

- *AQtime standalone:*
  - Select **Setup** from the **View** menu.
  - Select **Setup** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.
  - Select **Setup** from the **Assistant** panel.
- *AQtime integrated into Visual Studio:*
  - Select **Setup** from the **Select Panel** dialog, which is called by selecting **Profile | Panel List**.
  - Select **Setup** from Visual Studio's **Solution Explorer**.
  - Select **Setup** from the **Assistant** panel.
- *AQtime integrated into Borland Developer Studio:*
  - Double-click the needed AQtime project in Borland Developer Studio's **Project Manager**.
  - Select **Setup** from the **Assistant** panel.

The panel consists of three panes. Here is a sample:



Modules pane

The **Modules** pane is on the left of the Setup panel. It displays a list of all executable (exe and dll), assembly and script files available for profiling, in treelike format. Click on a module to view all namespaces, classes and methods within it. You can use the **View by** box that is located at the top of the Modules pane, to arrange information within the list by one of the following criteria:

Criterion	Description
Namespace	The module's routines are arranged by namespaces and then by classes. Note that if you select this value, the native-code modules will be empty, since native-code applications do not contain namespaces.
Class	The module's routines are arranged by class names.
Routine	The list holds only module's routines. Routines are displayed in the format *class_name.routine_name*.
Source File	The module's routines are arranged first by source file names and then by class names.
Unit	The module's routines are arranged first by unit name and then by class names. If you select this value, the .NET modules will be empty, since .NET does not use the term *unit*.
Default	This value selects optimal variant for each module in the Setup panel according to the module's "nature": suppose, you have a .NET and a native-code module in the Setup panel. If you select *Namespace* from the View by box, the native-code modules will appear empty. If you select *Unit*, the .NET module will be empty. The *Default* value lets you apply complex condition: the routines of .NET modules will be arranged by namespace, the routines from native-code modules will be arranged by units. That is, if you select *Default*, AQtime will apply the *Namespace* criterion for .NET modules and the *Unit* criterion for native-code modules.

To add a module (managed or unmanaged) to the project, select **Add Module** from the **Setup** toolbar or context menu. To add a .NET assembly to the project, select **Add Assembly**. To profile a script from a web page and add the page to the project, select **Add URL**. By default, the executable on which the project was first opened remains the "main"executable (it is displayed in bold). AQtime launches it when starting a profile run. Other modules will be loaded by this one as it runs (or possibly not loaded). If you want to make another module the "main"executable, select it and then choose **Set As Active Module** from the Setup toolbar or from the context menu. To remove the selected module from the project, select **Remove Module** from the Setup toolbar or context menu.

Alternatively, you can drag executables into the Modules pane from Windows Explorer. If they are dragged using the left button, they open a new project and become its main executable. If dragged with the right button, a dialog will pop up asking if you want to open a new project or add the executable as a module to the current project.

By default, AQtime profiles all the units that are supplied with the module including the modules that are embedded by IDEs, like MFC and VCL. However, generally, those units cannot be modified as their source code is not available, and therefore standard units are not worth profiling. To hide the standard units enable the *Exclude standard source files* option of the General Preferences dialog, or press the **Exclude Standard Source Files** button located on the Setup panel.

You can quickly locate a routine in the Modules pane without opening each module, by using **Find** on the context menu or on the **Edit** menu to call the Find dialog, or simply **Find Next**.

Areas pane

The **Areas** pane is on the right side, and at the top. It defines and keeps the list of profiling areas. Areas group code elements to be profiled (see *Defining Areas to Profile*). For an element to be profiled in a given run, it must be checked and its area must be checked also. In this way, areas do not just define code to profile, they keep definitions on hand for later use. There are also *excluding areas*, which subtract elements from larger blocks to profile (for example, one method from a class).

The pane displays areas, and each area can be opened to list its elements. There are two preset including areas, which you cannot change, remove or add elements to. The first, *Profile Entire .NET Code*, makes AQtime profile all functions that your application calls from any managed module (even if these functions do not belong to modules added to the **Setup** panel). The second preset area, *Full Check*, includes everything in the modules added to the Setup panel (it has effect if *Profile Entire .NET Code* is disabled). The third preset area, *Profile Entire Script Code*, allows you to profile scripts that are executed within the host application launched by AQtime. If none of preset areas is enabled, the other areas will be taken into account during profiling. Any of the preset areas (*Profile Entire .NET Code, Full Check* or *Profile Entire Script Code*) let you select the level, at which AQtime will profile your module(s): *routine, line* or *class*. For instance, if you select *Full Check by Lines*, AQtime will profile all modules at line level. For more information, see *Profiling Levels*.

Areas are like folders holding elements. You have first to add an area using **Add Area** from the **Setup** toolbar or context menu. The dialog lets you set the name for the area, and whether it will be *including* (default) or *excluding*. For the including area you can also specify its level: *class*, *routine* or *line* (see *Profiling Levels*). In the Setup panel, each area is displayed with the appropriate icon:

Areas are like folders holding elements. You have first to add an area using **Add Area** from the Setup toolbar or context menu. The dialog lets you set the name for the area, and whether it will be *including* (default) or *excluding*. For the including area you can also specify its level: *class*, *routine* or *line* (see *Profiling Levels*). In the Setup panel each area is displayed with the appropriate icon:

- Including Class Level area
- Including Routine Level area
- Including Line Level area
- Excluding Class Level area
- Excluding Routine Level area

You can change area settings later by using **Edit Area** on the context menu. This also has **Remove**.

Once you have an area defined you can add routines, classes, namespaces, units or modules to it. To do this, simply **drag the desired elements from the Modules pane to the area**. Another variant is to right-click the desired elements in the **Modules** pane and select the target area from the list in the **Add Selected to Area** submenu. You can also drag elements back out to the Modules pane, but using Remove is easier. A given element may belong to as many areas as you wish. If it is checked in an excluding area, however, this will override all checks in including areas.

If you check for inclusion some routines, but do not check the routines they call, then the execution time spend on those calls will be counted as if it belonged to the caller routine (see *Profiling Child Routines Along With Parents*). When you need to trace out exactly where the time goes, make sure that the child calls get profiled along with their callers. Triggers, described below, are an excellent tool for that.

Triggers and Actions pane

The **Triggers and Actions** pane is to the lower right of the Setup panel. Triggers and actions are organized in a fashion very similar to that of areas, but their purpose is different and they apply only to the **Performance** and **Coverage** profilers.

© 2010 AutomatedQA Corp.

Triggers are divided into *on-triggers* and *off-triggers*. In an on-trigger, whenever execution enters a checked element (routine, class, namespace), profiling is enabled for that thread. When execution leaves the element, profiling is returned to its former state. Vice-versa for off-triggers. For complete information on this, see *Using Triggers*.

Like triggers, actions can disable or enable profiling but unlike triggers actions do this for all the threads rather than for the current thread only. Another purpose of actions is to get profiling results during a profile run. Each action has effect either when execution enters a checked element (routine, class, namespace) or when execution leaves it. See *Using Actions*.

Note that profiling being enabled does not mean it actually operates. It means it is allowed to operate on the areas checked in the **Areas** pane. See *Controlling What to Profile*. However, you may simply check *Full Check* or *Profile Entire .NET Code* in the Areas pane, and then leave profiling control to triggers and actions.

There is one predefined trigger, **Initial Profiling Status for Threads**. It is checked by default. When it's checked, profiling is always enabled unless an off-trigger or an action that disables profiling is under execution. When this predefined trigger is unchecked, profiling is always disabled unless an on-trigger or an action that enables profiling is under execution. Leaving the predefined trigger checked means controlling profiling primarily through areas (which is always the case of course with all but the Performance profiler). To give primary control to triggers and actions, uncheck the predefined trigger.

To create a new trigger folder, select **Add Trigger** from the context menu. This will call the **Add Trigger** dialog, which allows you to specify the trigger name, type (on/off) and some other attributes (see *Setting up Triggers*).

To create a new action folder, select **Add Action** from the context menu. In the resulting **Action Properties** dialog, you will be able to specify the action name and attributes.

Once you have a trigger's or an action's folder, you add elements to it in the same way you would add them to an area (by dragging or by using the **Add Selected to Trigger** and **Add Selected to Action** items on the context menu). You can change trigger or action settings later by using **Edit** on the context menu. This also has **Remove**.

Setup Panel Options

The **Setup** panel offers an extensive complement of customizable options. You can check or change these options using the **Setup Options** dialog. To call it, do any of the following:

- *AQtime standalone:*
  - Select **Options** from the context menu of the Setup panel.
  - Select **Options | Options** from AQtime's main menu (this will call the Options dialog) and choose **Setup | Setup** from the tree view on the left of the dialog.
- *AQtime integrated into Visual Studio:*
  - Select **Options** from the context menu of the Setup panel.
  - Right-click **Setup** in Visual Studio's **Solution Explorer** and select **Properties** from the context menu.
  - Select **Tools | Options** from the main menu of Visual Studio and then choose the **AQtime | Setup | Setup** group in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Options** from the context menu of the Setup panel.
  - Select **Profile | Options** from the main menu of Borland Developer Studio (this will call the Options dialog) and choose **Setup | Setup** from the tree view on the left of the dialog.

The following options are available:

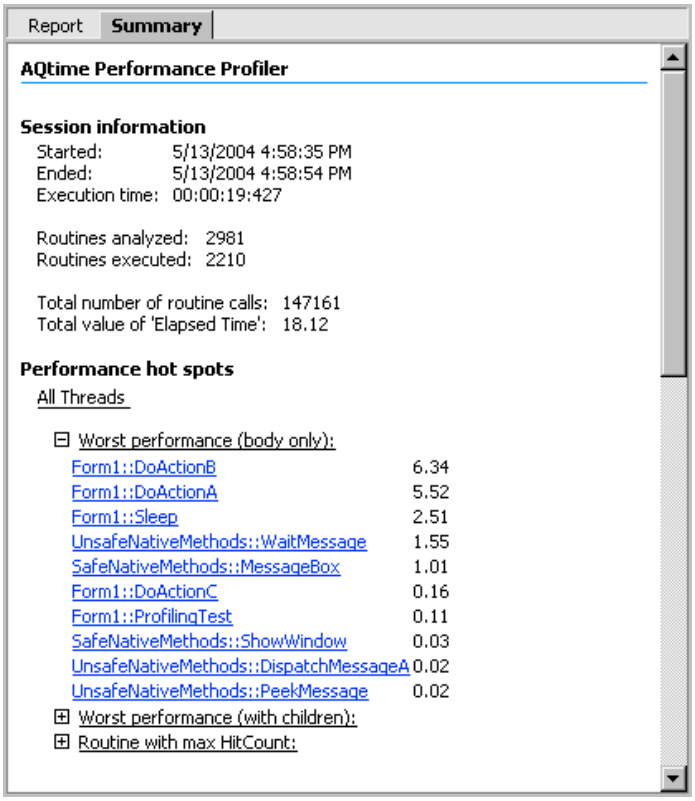
- **Activate after loading** - Sets whether AQtime will switch to the Setup panel after reading the debug info for the application, once AQtime finishes loading a project. Enabled by default.
- **Auto-select new elements** - Sets whether new areas, new triggers and new elements added to areas or triggers will be automatically checked when being added to the panel. If the option is on, the elements will be checked. Else - unchecked.

Summary Panel

The **Summary** panel displays brief profiling results for the result set that is currently selected in the **Explorer** panel. To display the Summary panel, do any of the following:

- *AQtime standalone:*
  - Select **Summary** from the **View** menu.
  - Select **Summary** from the **Select Panel** dialog, which is called by selecting **View | Select Panel**.
  - Select **Summary** from the **Assistant** panel.
- *AQtime integrated into Visual Studio:*
  - Select **Summary** from the **Select Panel** dialog, which is called by selecting **Profile | Panel List**.
  - Select **Summary** from Visual Studio's **Solution Explorer**.
  - Select **Summary** from the **Assistant** panel.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Summary** from the **View | Profile Windows** menu.
  - Select **Summary** from the **Assistant** panel.

Here is a sample view of the Summary panel:



The contents of the Summary panel depend on the current profiler, for example:

- For **Allocation** profiler results, the Summary panel reports about classes that are used the most and least often, namely:
  - Name of the class with maximum number of existing instances,
  - Name of the class with peak number of created instances,
  - The number of existing instances of classes that were included in profiling tasks,
  - The number of total created objects,
  - Etc.
- For **Performance** profiler results, the Summary panel contains information about routines that are performing poorly. This information depends on the counter you used to obtain the results. The Summary panel shows:
  - Routines with maximum Hit Count value,
  - Routines with top values in the Time (Misses, Branches, etc.) columns,
  - Routines with top values in the Time with Children (Misses with Children, Branches with Children, etc.) columns,
  - Etc.

Besides the profiler-dependant information, the Summary panel shows the time of the profiler run, information about profiler options, operating system, CPU, memory, etc.

The **Summary** clearly pin-points the sections of code that need to be optimized. This frees you from searching for this code in the profiling results yourself. To view detailed information on a routine or class that is shown in the Summary panel, simply click this routine or class in the panel. AQtime will then update its panels (for example, Report or Details) so that they display information regarding the routine or class you have clicked on.

AQtime Profilers

Performance Profiler

Performance Profiler - Overview

The **Performance** profiler is the next generation of AQtime's Timing profiler. It is your primary tool for investigating the performance of your 32-bit and 64-bit applications. It monitors the application execution and gathers loads of information about each application function, for instance, the number of function calls, the hierarchy of function calls, time spent executing the function, etc. It also provides information on pure .NET characteristics, such as time spent on JIT compilation and garbage collection (more below). The complete profiler description includes the following topics:

- *Overview (this topic)*
- *Supported Processor Models*
- *Analyzing Performance Profiler Results*
- *Description of the Report Panel Columns*
- *Description of the Details Panel Columns*
- *Performance Profiler Options*

Overview

According to its name, the **Performance** profiler serves for analyzing the application's performance. This profiler monitors all function calls in your application and measures different characteristics of the application. For instance:

- The time spent for executing a routine,
- The number of routine calls,
- The hierarchy of function calls,
- The number of CPU cache updates that occurred during the routine execution,
- And others.

What characteristic the profiler measures depends on the *Active Counter* profiler option (which you can change in the **Performance Profiler Settings** dialog that appears once this profiler starts). The following counters are available in the current AQtime version:

- *Elapsed Time*
- *User Time*
- *User+Kernel Time*
- *CPU Mispredicted Branches*

- *CPU Cache Misses*
- *Context Switches*
- *64K Aliasing Conflicts Split Load Replays*
- *Split Store Replays*
- *Blocked Store Forwards Replays*
- *Soft Memory Page Faults*
- *Hard Memory Page Faults*
- *All Memory Page Faults*

All the counters work both for managed and unmanaged code and support both 32bit and 64bit applications. For a complete description of counters, see *Counters Overview*.

Note that some counters may be unavailable. This depends on the CPU model and the software used. For instance, some counters do not work on Pentium II or do not support the processor's SpeedStep technology, while others do not function under virtual machines. Also, if you run AQtime x86 on a 64-bit operating system, the only available counter is *Elapsed Time*. For complete information on known counter restrictions, see *Counters Overview*.

Also, if you have Windows DDK installed, using some counters may cause the operating system to stop unexpectedly and display the error description on a blue screen. For more information on this problem and on how to solve it, see *Counters - Overview*.

Why do you need several counters? Because they help you **not only find** poorly performing functions, but determine **why** these functions are performing poorly during the profiler run. Suppose, you analyzed your application with the *Elapsed Time* counter and found that the FuncA routine runs too slow. This bottleneck can occur for several reasons:

- FuncA was called too many times;
- FuncA worked fast itself, but it called a slow child routine;
- FuncA called a routine from a system library or a .NET assembly that, in turn, took to much time to execute;
- If FuncA works with data in memory, the algorithm of its functioning might not be optimal so the CPU had to update its cache too many times during the function execution or too many hard page faults occurred;
- etc.

To determine the exact cause of poor application performance, you can profile FuncA and other slow routines with another counter and try to eliminate the bottleneck cause, if possible. For more information, *see Searching for Bottleneck Reasons With the Performance Profiler*.

Note: If you use a computer that has several processors or a multiple-core processor (for example, dual-core CPU) and has Windows XP Service Pack 2, then you must install the Windows update #896256 in order for the profiler to be able to time your application correctly. The update is available on Microsoft's web site:

- http://support.microsoft.com/kb/896256

No matter what counter you use, the Performance profiler lets you get static and runtime information concerning *Just-In-Time* (JIT) compilation of all methods in your .NET application. This includes, for example, the time spent for "JITting" (JIT compilation) or the number of CPU cache updates that occurred during JITting. The profiler works by logging special notification events generated by the JITting and of each method. One use of the profiler is to find the compiling-time (JITting-time) cost for your methods. You can then work to simplify those that waste the most time.

The profiler also collects such .NET-specific information as the garbage-collection time for each profiled routine. The garbage collector pauses the .NET application and the time spent for the garbage collection is included in the function execution time. The Performance profiler lets you determine the portion of garbage collection time in the total function execution time. For more information about the JIT compilation and garbage collection times, see *<JIT compiler> and <Garbage collector> Routines*.

The Performance profiler analyzes the application code at two levels of detail: **routine** and **line**. To profile the lines of a routine, you should simply add this routine to a line-level area (see *Profiling Levels*). Note that to profile routines at line level, you have to compile the application with debug information. See *How AQtime Profilers Use Metadata and Debug Information*.

The profiler also supports triggers and actions. They allow you to turn the profiling on or off exactly when it is needed during the application run. For more information, see *Using Triggers* and *Using Actions*.

The results of the Performance profiler runs are displayed in the **Report**, **Details**, **Call Graph**, **Call Tree** and **Editor** panels. For more information, we refer you to *Analyzing Performance Profiler Results*.

One note at the end: when you profile applications in Windows NT, sometimes the profiled process may remain in the operating system after the application termination. The information about the process is not removed from the operating system, so it "thinks" the process still exists. This happens because of certain limitations of the operating system. To kill such non-existent processes, you should restart Windows NT.

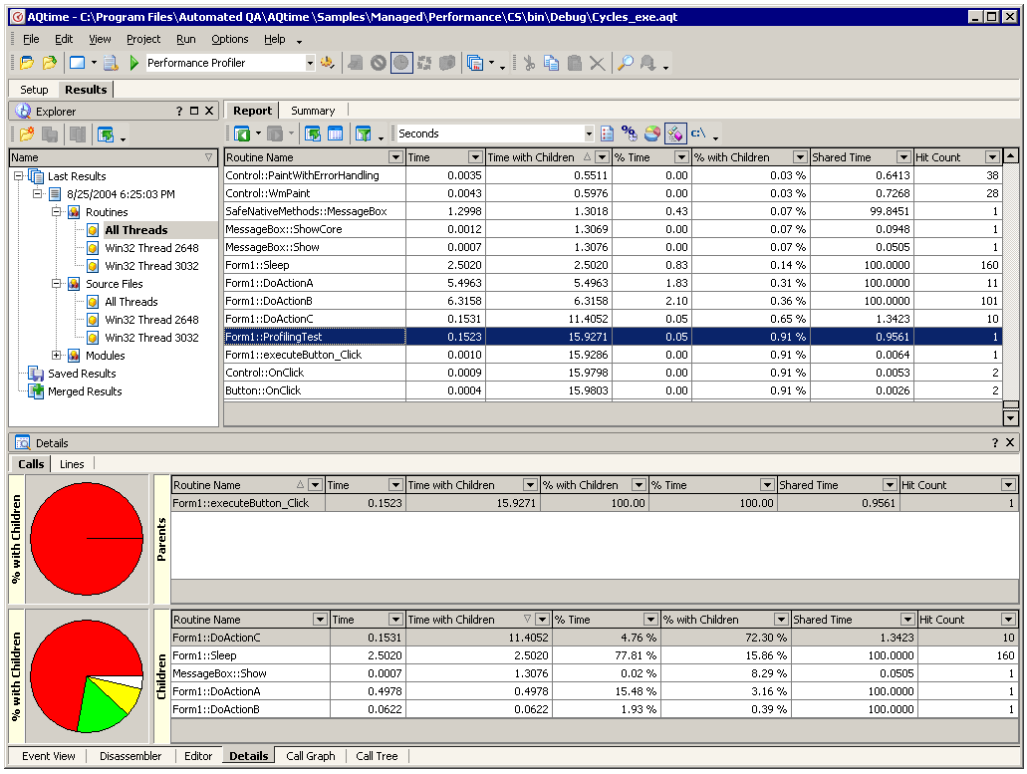
Analyzing Performance Profiler Results

The **Performance** profiler monitors all of the method calls in your application, counting the calls, tracing the call hierarchy (what called what), etc. thread by thread. The profiling results are organized into three categories: **Routines**, **Source Files** and **Modules**. *Source Files* and *Modules* let you view summary profiling results for each source file and module in your application. The *Routines* category contains results for each single routine that was included in profiling tasks.

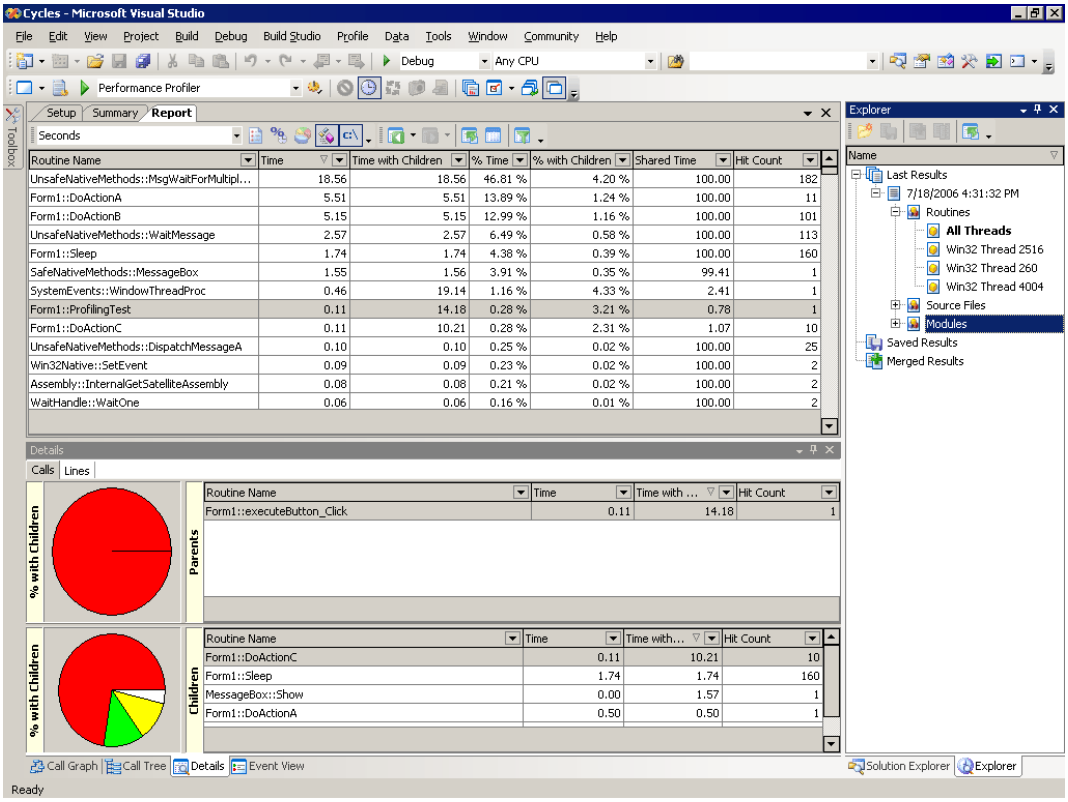
Within the categories the results are grouped by thread. There is also *All threads* group that shows profiling results for all threads.

Here is a sample output of the Performance profiler:

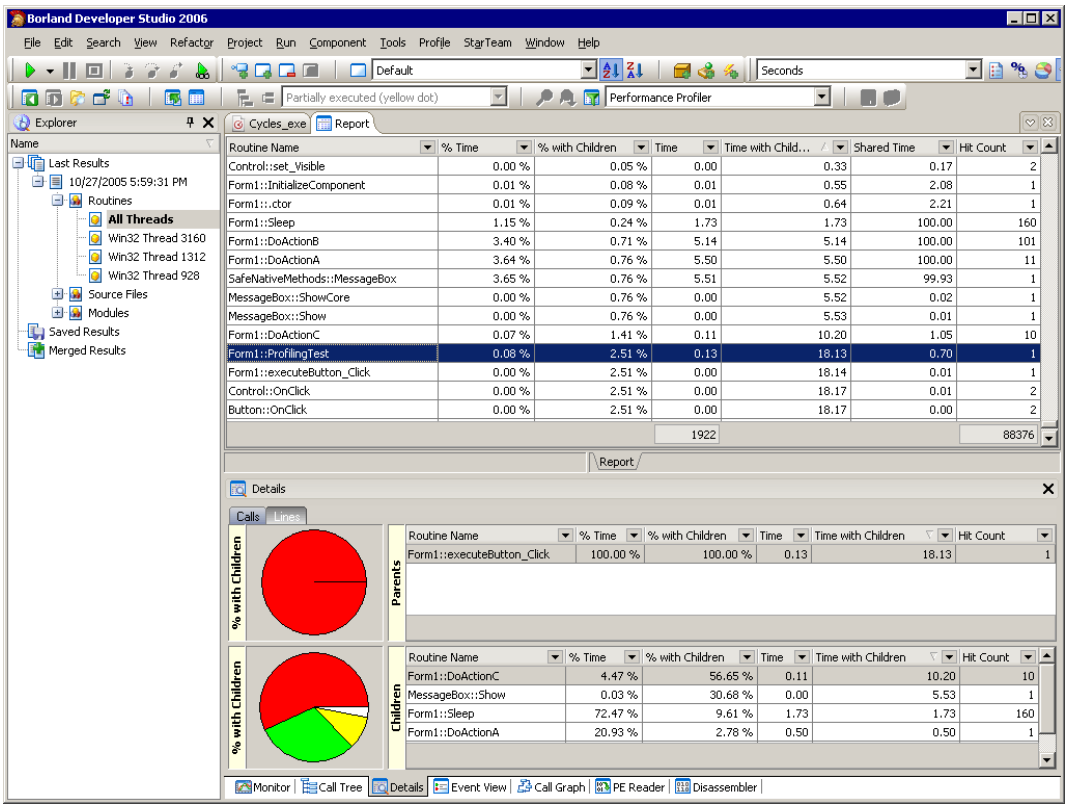
© 2010 AutomatedQA Corp.


AQtime - C:\Program Files\Automated QA\AQtime \Samples\Managed\Performance\CS\bin\Debug\Cycles_exe.aqt
File Edit View Project Run Options Help
Performance Profiler
Setup Results
Explorer
Report Summary
Seconds
Name
Last Results
8/25/2004 6:25:03 PM
Routines
All Threads
Win32 Thread 2648
Win32 Thread 3032
Source Files
All Threads
Win32 Thread 2648
Win32 Thread 3032
Modules
Saved Results
Merged Results
Routine Name | Time | Time with Children | % Time | % with Children | Shared Time | Hit Count
Control::PaintWithErrorHandling 0.0035 0.5511 0.00 0.03 % 0.6413 38
Control::WmPaint 0.0043 0.5976 0.00 0.03 % 0.7268 28
SafeNativeMethods::MessageBox 1.2998 1.3018 0.43 0.07 % 99.8451 1
MessageBox::ShowCore 0.0012 1.3069 0.00 0.07 % 0.0948 1
MessageBox::Show 0.0007 1.3076 0.00 0.07 % 0.0505 1
Form1::Sleep 2.5020 2.5020 0.83 0.14 % 100.0000 160
Form1::DoActionA 5.4963 5.4963 1.83 0.31 % 100.0000 11
Form1::DoActionB 6.3158 6.3158 2.10 0.36 % 100.0000 101
Form1::DoActionC 0.1531 11.4052 0.05 0.65 % 1.3423 10
Form1::ProfilingTest 0.1523 15.9271 0.05 0.91 % 0.9561 1
Form1::executeButton_Click 0.0010 15.9286 0.00 0.91 % 0.0064 1
Control::OnClick 0.0009 15.9798 0.00 0.91 % 0.0053 2
Button::OnClick 0.0004 15.9803 0.00 0.91 % 0.0026 2
Details
Calls Lines
% with Children
Parents
Form1::executeButton_Click 0.1523 15.9271 100.00 100.00 0.9561 1
% with Children
Children
Form1::DoActionC 0.1531 11.4052 4.76 % 72.30 % 1.3423 10
Form1::Sleep 2.5020 2.5020 77.81 % 15.86 % 100.0000 160
MessageBox::Show 0.0007 1.3076 0.02 % 8.29 % 0.0505 1
Form1::DoActionA 0.4978 0.4978 15.48 % 3.16 % 100.0000 1
Form1::DoActionB 0.0622 0.0622 1.93 % 0.39 % 100.0000 1
Event View Disassembler Editor Details Call Graph Call Tree

AQtime standalone

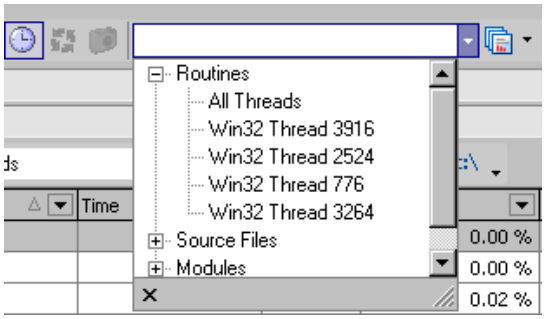

Cycles - Microsoft Visual Studio
File Edit View Project Build Debug Build Studio Profile Data Tools Window Community Help
Debug Any CPU
Performance Profiler
Setup Summary Report
Seconds
Routine Name | Time | Time with Children | % Time | % with Children | Shared Time | Hit Count
UnsafeNativeMethods::MsgWaitForMultipl... 18.56 18.56 46.81 % 4.20 % 100.00 182
Form1::DoActionA 5.51 5.51 13.89 % 1.24 % 100.00 11
Form1::DoActionB 5.15 5.15 12.99 % 1.16 % 100.00 101
UnsafeNativeMethods::WaitMessage 2.57 2.57 6.49 % 0.58 % 100.00 113
Form1::Sleep 1.74 1.74 4.38 % 0.39 % 100.00 160
SafeNativeMethods::MessageBox 1.55 1.56 3.91 % 0.35 % 99.41 1
SystemEvents::WindowThreadProc 0.46 19.14 1.16 % 4.33 % 2.41 1
Form1::ProfilingTest 0.11 14.18 0.28 % 3.21 % 0.78 1
Form1::DoActionC 0.11 10.21 0.28 % 2.31 % 1.07 10
UnsafeNativeMethods::DispatchMessageA 0.10 0.10 0.25 % 0.02 % 100.00 25
Win32Native::SetEvent 0.09 0.09 0.23 % 0.02 % 100.00 2
Assembly::InternalGetSatelliteAssembly 0.08 0.08 0.21 % 0.02 % 100.00 2
WaitHandle::WaitOne 0.06 0.06 0.16 % 0.01 % 100.00 2
Explorer
Name
Last Results
7/18/2006 4:31:32 PM
Routines
All Threads
Win32 Thread 2516
Win32 Thread 260
Win32 Thread 4004
Source Files
Modules
Saved Results
Merged Results
Details
Calls Lines
% with Children
Parents
Routine Name | Time | Time with ... | Hit Count
Form1::executeButton_Click 0.11 14.18 1
% with Children
Children
Routine Name | Time | Time with... | Hit Count
Form1::DoActionC 0.11 10.21 10
Form1::Sleep 1.74 1.74 160
MessageBox::Show 0.00 1.57 1
Form1::DoActionA 0.50 0.50 1
Call Graph Call Tree Details Event View
Solution Explorer Explorer
Ready

AQtime integrated into Visual Studio


Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Explorer
Cycles_exe
Report
Routine Name % Time % with Children Time Time with Child... Shared Time Hit Count
Control::set_Visible 0.00 % 0.05 % 0.00 0.33 0.17 2
Form1::InitializeComponent 0.01 % 0.08 % 0.01 0.55 2.08 1
Form1::.ctor 0.01 % 0.09 % 0.01 0.64 2.21 1
Form1::Sleep 1.15 % 0.24 % 1.73 1.73 100.00 160
Form1::DoActionB 3.40 % 0.71 % 5.14 5.14 100.00 101
Form1::DoActionA 3.64 % 0.76 % 5.50 5.50 100.00 11
SafeNativeMethods::MessageBox 3.65 % 0.76 % 5.51 5.52 99.93 1
MessageBox::ShowCore 0.00 % 0.76 % 0.00 5.52 0.02 1
MessageBox::Show 0.00 % 0.76 % 0.00 5.53 0.01 1
Form1::DoActionC 0.07 % 1.41 % 0.11 10.20 1.05 10
Form1::ProfilingTest 0.08 % 2.51 % 0.13 18.13 0.70 1
Form1::executeButton_Click 0.00 % 2.51 % 0.00 18.14 0.01 1
Control::OnClick 0.00 % 2.51 % 0.00 18.17 0.01 2
Button::OnClick 0.00 % 2.51 % 0.00 18.17 0.00 2
1922 88376
Details
Calls Lines
% with Children
Parents
Routine Name % Time % with Children Time Time with Children Hit Count
Form1::executeButton_Click 100.00 % 100.00 % 0.13 18.13 1
Children
Form1::DoActionC 4.47 % 56.65 % 0.11 10.20 10
MessageBox::Show 0.03 % 30.68 % 0.00 5.53 1
Form1::Sleep 72.47 % 9.61 % 1.73 1.73 160
Form1::DoActionA 20.93 % 2.78 % 0.50 0.50 1
Monitor Call Tree Details Event View Call Graph PE Reader Disassembler

AQtime integrated into Borland Developer Studio

As you can see, the categories and threads are shown in the **Explorer** panel. You can also select the desired category and thread using the Result Items toolbar item (by default, this item is hidden):



The **Summary** panel displays the summary results for the whole profiler run regardless of the selected category.Use this panel to quickly find routines that are performing poorly. The contents of other panels depend on the currently selected category:

- If you select the *Routines* category, AQtime will display profiling results one routine per line in the **Report** panel. Line timing results are displayed in the **Lines** page of the **Details** panel and in the **Editor**'s or **Code Editor**'s grid. The Report panel is the "main" results display. Other AQtime panels, such as Details, Call Graph or Call Tree, hold additional results for the routine selected in the Report panel.
- If you select the *Modules* or *Source Files* category, AQtime will display profiling results one module (or source file) per line in the Report panel. The other panels that provide additional information on profiling results (Details, Call Graph, etc.) are not used.

Results displayed in AQtime panels depend on the counter that was in use during the profiling run. For instance, if you profiled your application with the *Elapsed Time*, *User Time* or *User+Kernel Time* counters, AQtime panels will hold timing results. In further explanations we will mention results of the time counters only. Results of the other counters can be interpreted similar to the timing results and you can work with them in the same manner as you work with the timing results.

Profiling Results - Report Panel

The **Report** panel displays results for the category and thread that is selected in the Explorer panel or in the **Result Items** box on the Standard toolbar.

As we mentioned above, the names and values of the Report panel columns depend on the counter you used to profile your application. For more information about the available columns, see *Performance Profiler - Report Panel Columns*. Note that by default the Report panel shows only a shred of available columns. You can easily add more columns to the panel. For more information on this, see *Adding and Removing Columns*. You can arrange the columns in the panel as you desire: move columns, change column width, etc. For more information on this, see *Arranging Columns, Lines and Panels*.

The footer of the Report panel column holds summary values for data displayed in that column. For instance, the footer of the Hit Count column displays the total number of calls of all profiled methods. If you select two or more routines in the Report panel, the footer will show the summary values for the selected routine only (for more information on how to select several rows in a panel, see *Selecting Several Records in a Panel*).

The **Profiler** toolbar contains items that allow you to modify the results that are currently being displayed as your needs dictate. For example, if you use the *Elapsed Time*, *User Time* or *User+Kernel Time* counter, the *Counter unit* box lets you select the unit of time measurement for "time" columns. Another toolbar item, *Show non-hit routines*, lets you easily include or exclude non-executed routines from the result display. For more information on the toolbar items, see *Performance Profiler - Options*.

The column footer shows summary results for the values displayed in that column. You can customize the summary type and summary format using the **Format Columns** dialog. For instance, you can select one of the five summary types (Sum, Count, Avg, Min, Max) or you can hide the summary for the column.

To find the slowest routines, select the *Routines* category and then sort results by the **Time** (**Time with Children**) or **% Time** (**% with Children**) column. There are two predefined result views for the Performance profiler: **More than 3% (body only)** and **More than 3% (with children)**. They filter results to display only those routines that take the most time to execute their own code (for example, **% Time** is greater than 3) or their own code along with the code of all other routines they call (**% with Children** is greater than 3). To hide the **Time** and **Time with Children** columns and to display the columns **% Time** and **% with Children** instead, use another predefined result view of the Performance profiler: **Default with '%' fields**. (This view touches not only Time columns, but also other similar columns, for example, **Faults** and **% Faults**, **Branches** and **% Branches**, etc.)

You can select any of these result views from the **Result Views** dropdown list on the Standard toolbar, from the **View | Result Views** menu or from the **Result Views** dialog. To display it, choose **Profile | Result Views** from Visual Studio's menu or click **Result Views** on Borland Developer Studio's **View.AQtime** toolbar. See *Result Views*.

You can also group results by any column. When you group results by a column, besides "global" summaries shown at the footer of the panel, AQtime displays "local" summaries at the end of each group node. For instance, grouping results by the Class column helps you find the total time spent executing all class methods (the summary for the Time column should be enabled). For more information on how to group, sort, filter and search profiling results, see *Analyzing Profiler Results*.

Make sure that you compare the columns in the Report panel. Most of the methods within the profiled application call other ones. A fast method can call a slow one, making the caller appear slow too. If there is a big difference between Time and Time with Children columns (or % Time and % with Children columns), then the child methods slow down the method analyzed on that line.

The usefulness of the **% with Children** column is that it tells you which calls are the expensive calls. A function may cost time due to its own code, due to the child calls it makes, or due to the time spent on the JIT compiling or on the garbage collecting - but in any case it costs time. Often, an optimization will consist simply of making more efficient child calls - for instance, in moving a child call out of a loop. **% Time** reports the cost of the function's own code. % Time with Children reports the actual cost of running the function, no matter if the cost is incurred in the function's code or in the calls it makes.

The *% with children relative to real values* option does not change the relationship between the values in this column; the longest remains the longest and what is half as long remains half as long. If the option is enabled, the figures are all simply made larger (and the column total is above 100%). With % with children relative to real time enabled, 25% means that calls to the current function (and child calls) consumed a quarter of the entire profiled time. With the option disabled, the 25% would be much smaller, say 7.9%, and it would mean that calls to the current function (and child calls) consumed nearly 8% of the total time spent on any call during profiling, child calls being counted once for themselves, and again for their caller, and again for their caller's caller, etc. The column total would be 100%. See *Calculating Percent in the Report Panel* for more information.

Calls to child functions are timed (and deducted from the Time total) only if the child functions are part of the profiling areas. Otherwise, they count in the execution time of the parent function ("own code"). You may misidentify bottlenecks unless you make sure that the child functions are profiled along with their parents (callers). Triggers may help you do this without profiling everything during the run. See also *Profiling Child Routines Along With Parents*.

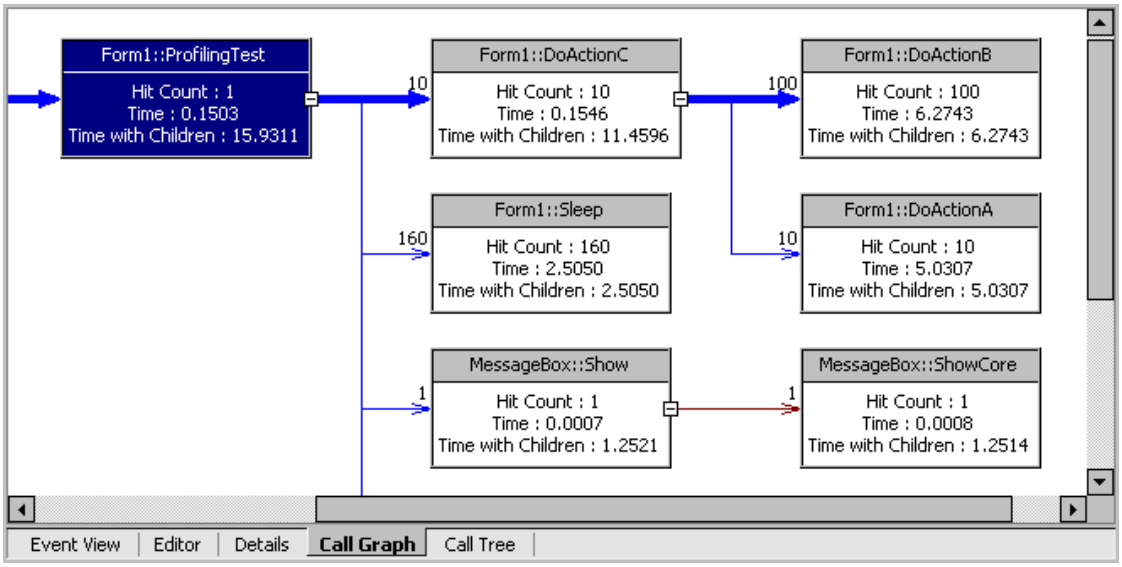
The Performance profiler results may include the <*Root*>, <*JIT compiler*> and <*Garbage collector*> pseudo-routines. These are fictitious routines, they do not exist in your application. <Root> is used as a parent routine of the topmost level; <JIT compiler> and <Garbage collector> help you figure out the time spent on the JIT compilation and garbage collection. See <*Root*> *Routine* and <*JIT compiler*> *and* <*Garbage collector*> *Routines* for more information about these functions.

For more information on how to compare and merge results of several profiler runs, see *Comparing and Merging Results*.

Profiling Results - Call Graph, Call Tree, Editor and Details Panels

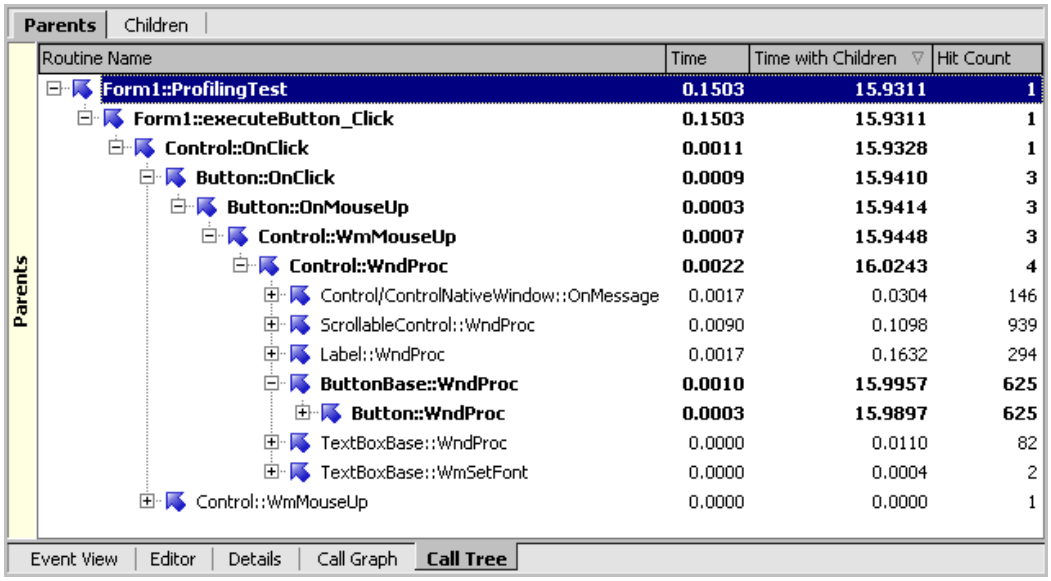
The **Performance** profiler displays profiling results in the **Call Graph**, **Call Tree**, **Details** and **Editor** panels when you view results of the *Routines* category. When you double-click on a routine in **Report**, AQtime refreshes these panels so that they will display information about this routine (see *AQtime Panels*.)

The **Call Graph** displays the function calls hierarchy (parent - child), centering on the clicked method. You can travel up and down the call tree in the panel by clicking the routines' rectangles.

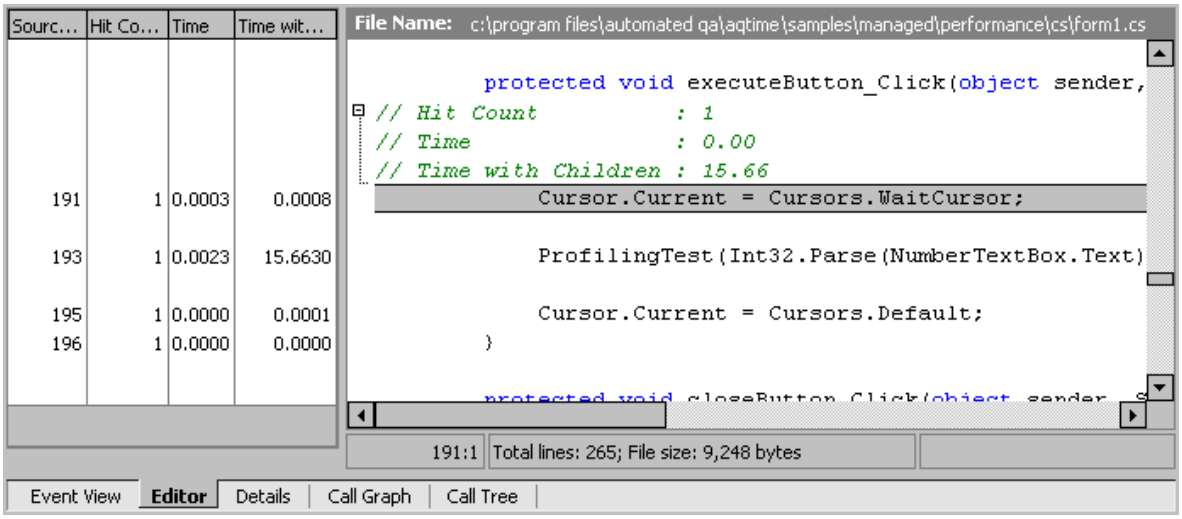


The critical path for the routine is displayed in bold (*critical path* is the "longest" path for the routine in the hierarchy of function calls, for instance, the one that took the longest to execute).

The **Call Tree** panel also displays the hierarchy of function calls. It contains two tables: Parents and Children. The **Parents** table holds all function calls that lead to the call to the currently selected routine. The **Children** table displays the hierarchy of old child calls that were initiated from the selected routine.



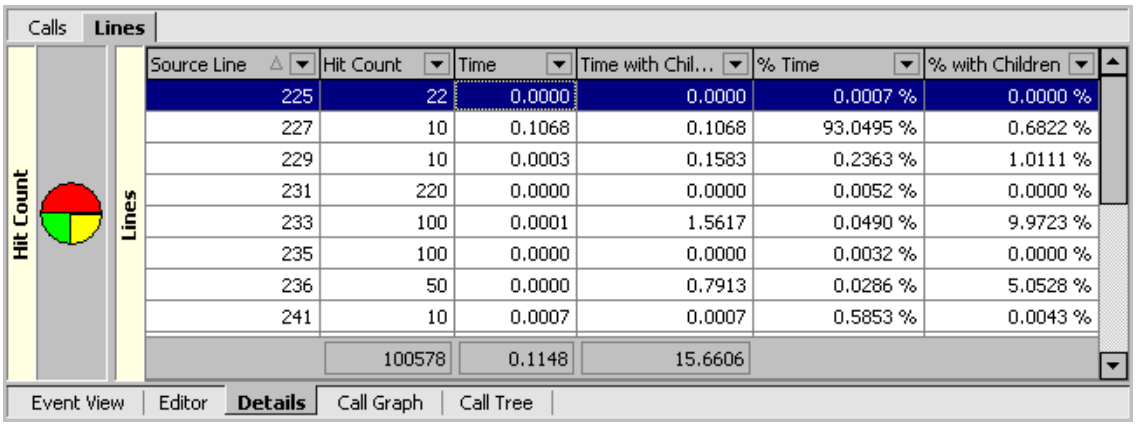
If your application was compiled with debug info, the **Editor** panel will also show the source code for the routine you clicked (The source file of the routine must be specified in the *Project Search Directories*. See also *How AQtime Profilers Use Metadata and Debug Information*). The Editor gutter displays various profiling results (Time, Hit Count, etc.) next to each source code line. To select which columns to display in the gutter, use the Field Chooser window. To bring it up, select **Field Chooser** from the context menu. See *Adding and Removing Columns*. The line profiling results are available, of course, only if the routine was profiled at line level:



Note for users who work in Visual Studio: The Code Editor of Visual Studio lets you collapse and expand blocks of source code. The grid, which AQtime adds to the Code Editor to display profiling results, supports neither collapsing, nor expanding, because Visual Studio does not send appropriate notifications to AQtime. So, to ensure that the grid shows proper profiling results for source lines and routines, please expand all the collapsed blocks of code. To do this, use the **Outlining | Toggle All Outlining** or **Outlining | Stop Outlining** item of the Code Editor's context menu.

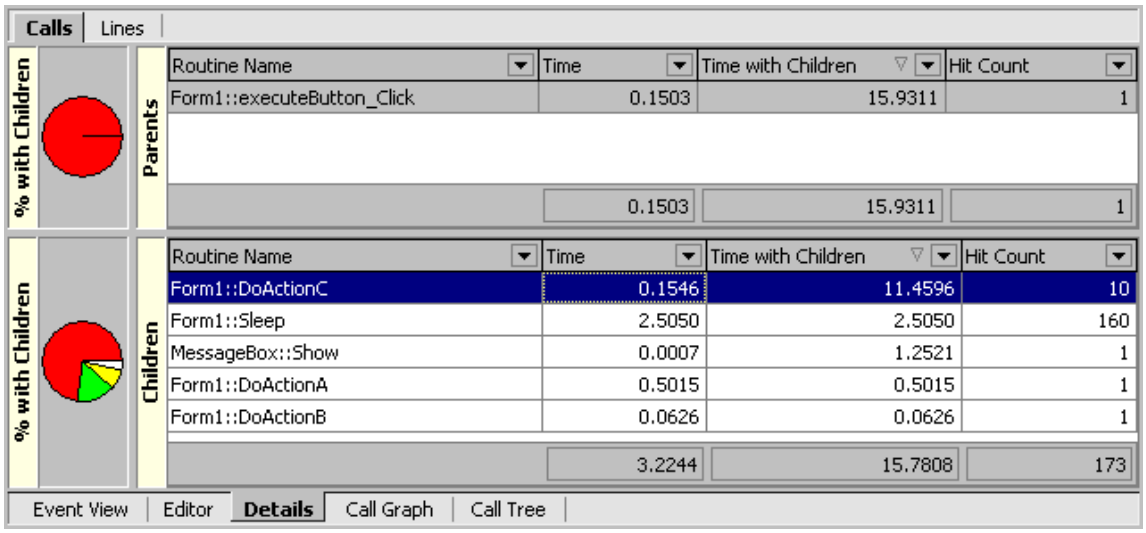
Note for users who work in Borland Developer Studio: The Editor of Borland Developer Studio lets you collapse and expand blocks of source code. The grid, which AQtime adds to the Editor to display profiling results, supports neither collapsing, nor expanding, because Borland Developer Studio does not send appropriate notifications to AQtime. So, to ensure that the grid shows proper profiling results for source lines and routines, please expand all the collapsed blocks of code. To do this, use the **Unfold | All** item of the Editor's context menu.

The line profiling results are also displayed in the **Lines** page of the **Details** panel. To view them, select the desired routine in the Report panel and then switch to Details:



The Lines page is very similar to the Report panel. To view the line in the Editor panel, simply double-click that line in the Lines table.

Another page of the Details panel, **Calls**, acts as a "magnifier" for parent-child call relationships related to one row in the Report panel. Here is an example:



Some results in the **Parents** and **Children** tables belong to the routine currently selected in the Report panel. For instance, the **Time** column in Parents displays the time spent by the selected routine in its parent routine; the **Time** column in Children displays the time spent by a child routine in the selected routine. For more information on results displayed in the Details panel as well as for the column description, see *Performance Profiler - Details Panel Columns*.

Note on percent columns: The columns that display percent values in the Children table (% Time, % with Children, % Branches, %Misses, % page Faults and others) depend on the *Include routine body in Details* setting that is shown in the toolbar of the Report panel. When this setting is enabled, AQtime displays information on the routine body execution in the Children table. This changes the number of rows in the table, which, in turn, changes the percent values.

Double-clicking on a row in the Details panel will open the Editor and position the cursor on the routine that was clicked. The double-click will also update the other panels to the routine displayed on that row. Switching from panel to panel in this way, when trying to get the desired information out of the Performance profiler results, is made much easier by the "browser" buttons, **Display Previous** and **Display Next**, on the Report toolbar.

You can arrange the Lines, Parents and Children tables within the Details panel as you desire. For more information on this, see description of the Details panel.

Calculating Percent in the Report Panel

The **Performance** profiler includes a *Calculate % with children relative to real values* option that is displayed on the **Profiler** toolbar. This option affects how the **% with children** values are calculated in the Report panel.

This option was primarily designed for the timing counters (*Elapsed Time*, *User Time* and *User+Kernel Time*). That is why in further explanations we will only mention the "Time" columns. However, the description can be extended on other counters, since their results are similar to the results of the timing counters.

In the default state the option is disabled, so **% Time** is the **Time** value as a percentage of the total of all Time values (as shown in the footer of the Time column). Likewise **% with children** is the **Time with Children** value as a percentage of the total of all values in that column. The total in the footer of % with children is 100%. For instance:

Report

Routine Name	Time	% Time	Time with Children	% with Children
Form1::.ctor	0,11	0,51 %	0,91	1,15 %
Form1::closeButton_Click	0,02	0,10 %	0,06	0,07 %
Form1::Dispose	0,04	0,17 %	0,04	0,05 %
Form1::DoActionA	5,54	25,22 %	5,54	6,98 %
Form1::DoActionB	5,10	23,21 %	5,10	6,43 %
Form1::DoActionC	0,10	0,45 %	10,20	12,85 %
Form1::executeButton_Click	0,01	0,03 %	16,52	20,82 %
Form1::InitializeComponent	0,80	3,64 %	0,80	1,01 %
Form1::Main	4,49	20,41 %	21,97	27,69 %
	21,97	100,00 %	79,35	100,00 %

When the option is enabled (the box is pressed), **% with children** is the **Time with Children** value as a percentage of the total of all values in the **Time** (not **Time with children**) column. Normally, this will yield a total in the footer of the % with children column much greater than 100%, as child time is being added in more than once (see the image below). The advantage of the setting is that % with children tells you what any profiled function costs, child calls included, as a proportion of the total profiled time. That is, for timing counters, “% with children relative to real values” is shorthand for “% relative to total time spent profiling”.

Report

Routine Name	Time	% Time	Time with Children	% with Children
Form1::.ctor	0,11	0,51 %	0,91	4,15 %
Form1::closeButton_Click	0,02	0,10 %	0,06	0,27 %
Form1::Dispose	0,04	0,17 %	0,04	0,17 %
Form1::DoActionA	5,54	25,22 %	5,54	25,22 %
Form1::DoActionB	5,10	23,21 %	5,10	23,21 %
Form1::DoActionC	0,10	0,45 %	10,20	46,40 %
Form1::executeButton_Click	0,01	0,03 %	16,52	75,17 %
Form1::InitializeComponent	0,80	3,64 %	0,80	3,64 %
Form1::Main	4,49	20,41 %	21,97	100,00 %
	21,97	100,00 %	79,35	361,10 %

When results are stored, they are stored with the current column contents. % with children will not change when you retrieve the results, no matter what the current setting for % with children relative to real values. You can easily see under what setting the results were generated by checking the footer for % with children. If it is 100%, then the option was off. If it is greater, the option was on.

<JIT compiler> and *<Garbage collector>* Routines

The total execution time of a managed (.NET) routine includes the following four times:

- Time spent for executing the routine's own code.
- Time spent for running the child routines.
- Time spent by the Just-In-Time compiler (JIT compiler) for compiling the routine's code: before executing a routine, the Common Language Runtime (CLR) may compile this routine at run time. Compilation takes some time, this time is included in the total execution time of the routine. AQtime traces the compilation requests and calculates JIT compilation time.

© 2010 AutomatedQA Corp.

- Time spent for the garbage collecting: the CLR pauses the .NET application, when the garbage collector starts, and resumes the application after the garbage collection is over. The time spent for garbage collecting is included in the total execution time of the routine.

To help you find out how much time was spent on the Just-In-Time compiling and garbage collecting during the application run, the **Performance** profiler includes the *Profile .NET runtime* option. If this option is enabled, AQtime times the JIT compilation and garbage collection routines and display them as <*JIT compiler*> and <*Garbage collector*> in the **Report** and **Details** panels. Note that both <JIT compiler> and <Garbage collector> are fictitious routines. They do not exist in your application. Below is a description of the profiling results in the Report and Details panels.

One note before we proceed: information the profiler gathers for the <*JIT compiler*> and <*Garbage collector*> routines depends on the active counter. Most likely you will use "time" counters, therefore, any further explanations will only mention the "Time" columns. However, this does not mean you cannot use other counters. The results other counters produce are similar to the results of the "time" counters. Replacing "Time" with the appropriate term (for example, "Misses") in the following paragraphs will keep the description valid:

- **Report panel**. The **Time** column for the <*JIT compiler*> routine in the Report panel displays the total time spent by the JIT compiler for compiling application routines during the run. The Time column for the <*Garbage collector*> routine in the Report panel shows the total time spent by the CLR for garbage collecting during the application run.

  Since both <*JIT compiler*> and <*Garbage collector*> are fictitious routines, they do not have child routines. That is why, the **Time with Children** result is equal to **Time**, and **% with Children** is equal to **% Time**.

- **Details panel**. The <*JIT compiler*> and <*Garbage collector*> routines are displayed in the Children table of the Details panel. This means that they were called from the routine, which is currently selected in the Report panel. They let you determine what portion of time was included in the total execution time of the routine, but was not spent executing the routine's own code or for child calls:

  - The **Time** column of the <*JIT compiler*> routine shows the time spent by the JIT compiler for compiling the routine.
  - The **Time** column of the <*Garbage collector*> function indicates the time spent on garbage collecting.

  Like in the Report panel, the Details' **Time with Children** value is equal to **Time** and **% with Children** is equal to **% Time**.

  To see the <*JIT compiler*> and <*Garbage collector*> results for a routine, click this routine in the Report panel and then switch to the Children pane of the Details panel. The chart on the left side of the Children pane graphically displays what portion of time was taken by the JIT compilation and garbage collection.

Results in the Details panel includes the <*JIT compiler*> and <*Garbage collector*> routines only if the Just-In-Time compiler or garbage collector was called during the routine execution. If the CLR did not run garbage collector during the routine execution, the routine's results in the Details panel will not contain the <*Garbage collector*> function.

The same applies to the <*JIT compiler*> routine. The CLR will not launch a JIT compiler, if a child function was compiled when the JIT compiler compiles a parent function. Therefore, the child function's results will not contain the <*JIT compiler*> routine. The time taken by the child function's JIT-compilation will be included in the <*JIT compiler*> result of the parent function. The <*JIT compiler*> function is also absent in the Details results, if the analyzed routine was pre-compiled (pre-JITted) before the application start and the CLR did not run the JIT compiler during the routine's execution.

Since the CLR may perform JIT compilation or garbage collection while executing native-code functions, <*JIT compiler*> and <*Garbage collector*> may appear in detailed results of native-code functions.

<Root> Routine

If the *Profile <Root> routine* option is enabled, the results of the **Performance** profiler include the **<*Root*>** routine. This is a hypothetic routine. It does not exist in your application. AQtime treats it as a parent routine of the topmost level. The <*Root*> body includes different initialization statements and function calls. All other routines are called from the <*Root*> one: the main routine, event handlers (for example, `OnClick`) that respond to user actions, etc.

Each application thread has its own <*Root*> routine. Even the thread function is "called" from <*Root*> (of course, this differs from what actually happens, because the first "parent" function in each thread is its thread function).

To view functions that are called from <*Root*>, click <*Root*> in the **Report** panel and switch to **Details**. The **Hit Count** column always displays 1 for the <*Root*> routine. As the <*Root*> body includes mostly initialization statements, the **Time** column for this routine displays the time needed for application initialization. **Time with Children** displays the overall time of the application run. Note that this time is less than the one you can see in the **Event View** panel. This happens because Event View displays the total time of the application run, which includes time taken by AQtime for operation profiling. The Report panel displays the "net" application time (without AQtime time).

Performance Profiler - Report Panel Columns

When displaying results of the **Performance** profiler, each row in the **Report** panel holds the results for a routine, source file or module in your application. Which values are displayed depends on the category selected in the **Explorer** panel and on the counter that was used for profiling.

Routines Category

Columns independent of the active counter

Columns (in alphabetical order)	**Description**
Address	Routine's address in memory. This column is used for unmanaged (native-code) routines only.
Analysis Result	Specifies if the routine was instrumented or not. If the routine was instrumented, this column is empty. Otherwise, the column displays a short description why the routine was not instrumented: • *Less than 5 bytes* - The routine occupies less than 5 bytes in memory. See *Profiling Small Functions*. • *No line info* - The routine was added to a line-level area, but the debug information holds no info about routine lines. These routines can be profiled at routine level only. • *Unsafe code* - AQtime was not able to instrument the routine safely. This typically occurs when the routine's binary code is intermixed with data areas. See *Profiling Routines That Hold Unsafe Code*. • *No ret instruction* - The routine's binary code does not contain the ret instruction (this may happen if the routine finishes with the jmp

	instruction). See *Profiling Routines That Do Not Have the ret Instruction*. • *Duplicated code* - The routine whose code coincides with code of another routine. To learn more about this, see *Profiling Duplicated Code*.
Class Name	Name of the class where the method is defined.
Code Type	Specifies the type of the routine's code. The following values are possible: • **MSIL** - Managed-code routine with MSIL (Microsoft Intermediate Language) code. • **x64** - 64-bit code routine. • **x86** - Native-code (unmanaged) routine. • **Pseudo** - Pseudo routine that was created by the context. For example, <JIT Compiler>, <Garbage Collector>, <Unknown PInvoke> or <Root>. • **PInvoke** - Native-code routine for which there is a declaration in one of managed modules and that is called from within the unmanaged code. • **NGen** - Managed routine that was compiled by the ngen utility (CLR Native Image Generator) with the /prof argument in its command line. The ngen compilation means the routine was compiled before the application starts. • **Script** - The routine belongs to a script that was profiled along with the host application. See *Profiling Scripts* for details.
Exceptions	Number of times the method was entered but not successfully exited. This is usually a count of exception exits.
Hit Count	The number of routine calls that were profiled. See also *Skip Count*. The total number of times the routine was executed is determined as *Hit Count + Skip Count*.
Max Recursion Depth	The maximum number of nested (recursive) calls to the function reached during the run. This value is calculated only if the *Track recursion depth* option is checked.
Module Name	The name of the module which contains the profiled routine.
Namespace	Namespace of the method's class (this column is used for managed routines only).
Routine Name	Method name.
Skip Count	Number of times the routine was excluded from profiling, because the profiling status was off (this can be, for example, the number of times the routine was affected by an off-trigger or the number of times the routine was executed when the Enable/Disable Profiling button was not pressed). See also *Hit Count*. The total number of times the routine was executed is determined as *Hit Count + Skip Count*.
Source File	Name of the source file for the method. The values for this column are read from the application's debug info.
Source Line	Source file's line number where the method's implementation begins. The

	values for this column are read from the application's debug info.
Token	The routine's token.
Unit Name	Name of the linkage unit that holds the routine. This column is used for unmanaged (native-code) routines only.

Columns that depend on the active counter

Elapsed Time, User Time and User+Kernel Time Counters

You can specify the measurement unit for the following columns (*seconds*, *milliseconds*, *microseconds* or *machine cycles*) using the *Counter unit* box on the **Profiler** toolbar. See also *Performance Profiler Options*.

Columns (in alphabetical order)	**Description**
Average Time	Average time spent executing the routine's code on one call. This is simply *Time / (Hit Count + Skip Count)*.
Average Time with Children	Average time spent on each call to the routine, child calls included. This is simply *Time with Children / (Hit Count + Skip Count)*.
First Time	Time spent executing the routine's code on the first call (child calls are excluded).
First Time With Children	Time spent executing the routine's code on the first call (including time of child calls that were made during the first routine's call). **Note:** The First Time and First Time With Children columns help you determine why a routine took too much time to execute: a routine itself can work quickly, but it can perform initialization actions on the first call. These actions may perform slowly and make the routine one of the slowest routines in your application. By comparing results in the First Time (First Time With Children) and Time (Time With Children) columns, you can determine where the bottleneck occurred.
Max Time **Min Time**	Maximum and minimum time spent executing the routine's code on a call. Exceptional values point out perhaps unexpected special conditions.
Max Time with Children **Min Time with Children**	Maximum and minimum time spent executing the routine's code on a call (including child function calls). Exceptional values point out perhaps unexpected special conditions.
Time	Total time spent executing the routine's code excluding child calls. The sum of all profiled methods appears in the footer of this column.
Time with Children	Total time spent on calls to the routine including calls to child routines. The sum for all profiled routines is displayed in the footer of this column. It will normally be an important multiple of actual profile-run duration, as child calls are counted several times.
Shared Time	Total time spent executing the routine’s code, as a percentage of the total time spent on calls to the routine including calls to child routines.This is simply (*Time* / *Time with Children*)**100*.
% Time	Total time spent executing the routine's code, as a percentage of the time spent executing all profiled routines.
% with Children	Time with Children value as a percentage of the sum of Time with Children for all profiled routines.

CPU Cache Misses Counter

Columns (in alphabetical order)	**Description**
Average Misses	Average number of cache misses that occurred during execution of the method's code on one call. This is simply *Misses / (Hit Count + Skip Count)*.
Average Misses with Children	Average number of cache misses that occurred during the routine execution (including cache misses that occurred in child calls). This is simply *Misses with Children / (Hit Count + Skip Count)*.
First Misses	The number of cache misses that occurred during execution of the function's code on the first call (child calls are excluded).
First Misses With Children	The number of cache misses that occurred during execution of the function's code on the first call (including child calls).
Max Misses **Min Misses**	Maximum and minimum number of cache misses that occurred during execution of the method's code on a call. Exceptional values point out perhaps unexpected special conditions.
Max Misses with Children **Min Misses with Children**	Maximum and minimum number of cache misses that occurred during the routine execution (including child function calls). Exceptional values point out perhaps unexpected special conditions.
Misses	Total number of cache misses that occurred during execution of the routine's code excluding child calls. The sum for all profiled routines appears in the footer of this column.
Misses with Children	Total number of cache misses that occurred during execution of the routine (including its calls to child methods). The sum for all profiled routines is displayed in the footer of this column.
Shared Misses	Total number of cache misses that occurred during the routine execution (excluding child calls), as a percentage of the total number of cache misses that occurred during the routine execution (including child calls). That is, this is simply (*Misses* / *Misses with Children*)**100*.
% Misses	Total number of cache misses that occurred during the routine execution (excluding child calls), as a percentage of the sum of cache misses that occurred during execution of all profiled routines.
% with Children	Total number of cache misses as a percentage of the sum of Misses with Children for all profiled routines.

CPU Mispredicted Branches Counter

Columns (in alphabetical order)	**Description**
Average Branches	Average number of branches that mispredicted during execution of the method's code on one call. This is simply *Branches / (Hit Count + Skip Count)*.
Average Branches with Children	Average number of branches that were mispredicted during the routine execution (including branches mispredicted in child calls). This is simply *Branches with Children / (Hit Count + Skip Count)*.
Branches	Total number of branches that were mispredicted during execution of the routine's code excluding child calls. The sum for all profiled

	routines appears in the footer of this column.
Branches with Children	Total number of branches that were mispredicted during execution of the routine (including mispredictions in child methods). The sum for all profiled routines is displayed in the footer of this column.
First Branches	The number of mispredictions that occurred during execution of the function's code on the first call (child calls are excluded).
First Branches With Children	The number of mispredictions that occurred during execution of the function's code on the first call (including child calls).
Max Branches **Min Branches**	Maximum and minimum number of mispredictions that occurred during execution of the method's code on a call. Exceptional values point out perhaps unexpected special conditions.
Max Branches with Children **Min Branches with Children**	Maximum and minimum number of mispredictions that occurred during the routine execution (including mispredictions in child functions). Exceptional values point out perhaps unexpected special conditions.
Shared Branches	Total number of mispredicted branches that occurred during the routine execution (excluding child calls), as a percentage of the total number of mispredictions that occurred during the routine execution (including child calls). That is, this is simply (*Branches / Branches with Children*)**100*.
% Branches	Total number of branches that were mispredicted during the routine execution (excluding child calls), as a percentage of the sum of mispredictions occurred during execution of all profiled routines.
% with Children	Total number of mispredicted branches as a percentage of the sum of Branches with Children for all profiled routines.

Hard Memory Page Faults, Soft Memory Page Faults and All Memory Page Faults Counters

Columns (in alphabetical order)	**Description**
Average Faults	Average number of page faults that occurred during execution of the method's code on one call. This is simply *Faults / (Hit Count + Skip Count)*.
Average Faults with Children	Average number of page faults that occurred during the routine execution (including page faults that occurred in child calls). This is simply *Faults with Children / (Hit Count + Skip Count)*.
Faults	Total number of page faults that occurred during execution of the routine's code (child calls are excluded). The sum for all profiled routines appears in the footer of this column.
Faults with Children	Total number of page faults that occurred during execution of the routine (including page faults that occurred in child methods). The sum for all profiled routines is displayed in the footer of this column.
First Faults	The number of page faults that occurred during execution of the function's code on the first call (child calls are excluded).
First Faults With Children	The number of page faults that occurred during execution of the

	function's code on the first call (child calls are included).
Max Faults **Min Faults**	Maximum and minimum number of page faults that occurred during execution of the method's code on a call. Exceptional values point out perhaps unexpected special conditions.
Max Faults with Children **Min Faults with Children**	Maximum and minimum number of page faults that occurred during the routine execution (including page faults that occurred in child functions). Exceptional values point out perhaps unexpected special conditions.
Shared Faults	Total number of page faults that occurred during the routine execution (excluding child calls), as a percentage of the total number of page faults that occurred during the routine execution (including child calls). That is, this is simply (*Faults* / *Faults with Children*)**100*.
% Faults	Total number of page faults that occurred during the routine execution (excluding child calls), as a percentage of the sum of page faults that occurred during execution of all profiled routines.
% with Children	Total number of page faults as a percentage of the sum of Faults with Children for all profiled routines.

Split Load Replays, Split Store Replays and Blocked Store Forwards Replays Counters

Columns (in alphabetical order)	**Description**
Average Replays	Average number of replays that occurred during execution of the method's code on one call. This is simply *Replays / (Hit Count + Skip Count)*.
Average Replays with Children	Average number of replays that occurred during the routine execution (including replays that occurred in child calls). This is simply *Replays with Children / (Hit Count + Skip Count)*.
First Replays	The number of replays that occurred during execution of the function's code on the first call (child calls are excluded).
First Replays With Children	The number of replays that occurred during execution of the function's code on the first call (child calls are included).
Max Replays **Min Replays**	Maximum and minimum number of replays that occurred during execution of the method's code on a call. Exceptional values point out perhaps unexpected special conditions.
Max Replays with Children **Min Replays with Children**	Maximum and minimum number of replays that occurred during the routine execution (including replays that occurred in child functions). Exceptional values point out perhaps unexpected special conditions.
Replays	Total number of replays that occurred during execution of the routine's code (child calls are excluded). The sum for all profiled routines appears in the footer of this column.
Replays with Children	Total number of replays that occurred during execution of the routine (including replays that occurred in child methods). The sum for all profiled routines is displayed in the footer of this column.
Shared Replays	Total number of replays that occurred during the routine execution (excluding child calls), as a percentage of the total number of

	replays that occurred during the routine execution (including child calls). That is, this is simply (*Replays* / *Replays with Children*)**100*.
% Replays	Total number of replays that occurred during the routine execution (excluding child calls), as a percentage of the sum of replays that occurred during execution of all profiled routines.
% with Children	Total number of replays as a percentage of the sum of Replays with Children for all profiled routines.

64K Aliasing Conflicts Counter

Columns (in alphabetical order)	**Description**
Average Conflicts	Average number of aliasing conflicts that occurred during execution of the method's code on one call. This is simply *Conflicts / (Hit Count + Skip Count)*.
Average Conflicts with Children	Average number of aliasing conflicts that occurred during the routine execution (including conflicts that occurred in child calls). This is simply *Conflicts with Children / (Hit Count + Skip Count)*.
Conflicts	Total number of aliasing conflicts that occurred during execution of the routine's code (child calls are excluded). The sum for all profiled routines appears in the footer of this column.
Conflicts with Children	Total number of aliasing conflicts that occurred during execution of the routine (including conflicts that occurred in child methods). The sum for all profiled routines is displayed in the footer of this column.
First Conflicts	The number of conflicts that occurred during execution of the function's code on the first call (child calls are excluded).
First Conflicts With Children	The number of aliasing conflicts that occurred during execution of the function's code on the first call (child calls are included).
Max Conflicts **Min Conflicts**	Maximum and minimum number of aliasing conflicts that occurred during execution of the method's code on a call. Exceptional values point out perhaps unexpected special conditions.
Max Conflicts with Children **Min Conflicts with Children**	Maximum and minimum number of aliasing conflicts that occurred during the routine execution (including conflicts that occurred in child functions). Exceptional values point out perhaps unexpected special conditions.
Shared Conflicts	Total number of aliasing conflicts that occurred during the routine execution (excluding child calls), as a percentage of the total number of aliasing conflicts that occurred during the routine execution (including child calls). That is, this is simply (*Conflicts* / *Conflicts with Children*)**100*.
% Conflicts	Total number of aliasing conflicts that occurred during the routine execution (excluding child calls), as a percentage of the sum of conflicts that occurred during execution of all profiled routines.
% with Children	Total number of aliasing conflicts as a percentage of the sum of Conflicts with Children for all profiled routines.

Context Switches Counter

Columns (in alphabetical order)	**Description**
Average Switches	Average number of context switches that occurred during execution of the method's code on one call. This is simply *Switches / (Hit Count + Skip Count)*.
Average Switches with Children	Average number of context switches that occurred during the routine execution (including switches that occurred in child calls). This is simply *Switches with Children / (Hit Count + Skip Count)*.
First Switches	The number of context switches that occurred during execution of the function's code on the first call (child calls are excluded).
First Switches With Children	The number of context switches that occurred during execution of the function's code on the first call (child calls are included).
Max Switches **Min Switches**	Maximum and minimum number of context switches that occurred during execution of the method's code on a call. Exceptional values point out perhaps unexpected special conditions.
Max Switches with Children **Min Switches with Children**	Maximum and minimum number of context switches that occurred during the routine execution (including switches that occurred in child functions). Exceptional values point out perhaps unexpected special conditions.
Switches	Total number of context switches that occurred during execution of the routine's code (child calls are excluded). The sum for all profiled routines appears in the footer of this column.
Switches with Children	Total number of context switches that occurred during execution of the routine (including switches that occurred in child methods). The sum for all profiled routines is displayed in the footer of this column.
Shared Switches	Total number of context switches that occurred during the routine execution (excluding child calls), as a percentage of the total number of context switches that occurred during the routine execution (including child calls). That is, this is simply (*Switches / Switches with Children*)**100*.
% Switches	Total number of context switches that occurred during the routine execution (excluding child calls), as a percentage of the sum of switches that occurred during execution of all profiled routines.
% with Children	Total number of context switches as a percentage of the sum of Switches with Children for all profiled routines.

Source Files and Modules Categories

Columns independent of the active counter

Column	**Description**
File Name or **Module Name**	Name of the source file (or module).
Exceptions	This value is a sum of all Exception values of routines that belong to the source file

	(module). It indicates the total number of times the methods belonging to the source file (or module) were entered but not successfully exited. This is usually the number of exception exits in the file's (module's) routines.
Hit Count	The number of routine calls that were profiled. See also *Skip Count*. This value is a sum of the Hit Count result of all profiled routines that belong to the given source file or module.
Skip Count	Number of times the routines were excluded from profiling, because the profiling status was off (This can be, for example, the number of times the routines were affected by off-triggers). This value is a sum of the Skip Count result of all profiled routines that belong to the given source file or module.

Columns that depend on the active counter

Column	Column	Description
Elapsed Time **User Time** **User+Kernel Time**	**Time**	The total execution time (excluding child calls) of routines that are defined in the source file (or module). This is a sum of the Time results of all profiled routines that the given source file (or module) contains.
	% Time	Total time spent for execution of profiled routines that belong to the given source file (module) as a percentage of the sum of the Time column values for all source files (modules) that are displayed in the Report panel.
CPU Cache Misses	**Misses**	The total number of CPU cache misses that occurred during profiling of routines that the given source file (or module) contains. This is a sum of the Misses result value of all routines that the given source file (or module) contains.
	% Misses	Total number of cache misses as a percentage of the sum of the Misses column values for all source files (modules) that are displayed in the Report panel.
CPU Mispredicted Branches	**Branches**	The total number of code branches that were mispredicted during profiling of routines that the given source file (or module) contains. This is a sum of the Branches result of all profiled routines that the given source file (or module) contains.
	% Branches	Total number of mispredicted branches as a percentage of the sum of the Branches column values for all source files (modules) that are displayed in the Report panel.
Hard Memory Page Faults **Soft Memory Page Faults** **All Memory Page Faults**	**Faults**	The total number of memory page faults that occurred during profiling of routines that the given source file (or module) contains. This is a sum of the Faults result value of all profiled routines that the given source file (or module) contains.
	% Faults	Total number of memory page faults as a percentage of the sum of the Faults column values for all source files (modules) that are displayed in the Report panel.

Split Load Replays, **Split Store Replays**, **Blocked Store Forwards Replays**	**Replays**	The total number of replays that occurred during profiling of routines that the given source file (or module) contains. This is a sum of the Replays result value of all profiled routines that the given source file (or module) contains.
	% Replays	Total number of replays as a percentage of the sum of the Replays column values for all source files (modules) that are displayed in the Report panel.
64K Aliasing Conflicts	**Conflicts**	The total number of aliasing conflicts that occurred during profiling of routines that the given source file (or module) contains. This is a sum of the Conflicts result value of all profiled routines that the given source file (or module) contains.
	% Conflicts	Total number of aliasing conflicts as a percentage of the sum of the Conflicts column values for all source files (modules) that are displayed in the Report panel.
Context Switches	**Switches**	The total number of context switches that occurred during profiling of routines that the given source file (or module) contains. This is a sum of the Switches result value of all profiled routines that the given source file (or module) contains.
	% Switches	Total number of context switches as a percentage of the sum of the Switches column values for all source files (modules) that are displayed in the Report panel.

Note that by default the Performance profiler shows a few of available columns in the **Report** panel. You can add more columns to the panel. For more information on this, see *Adding and Removing Columns*.

Performance Profiler - Details Panel Columns

When you view the **Performance** profiler results for your routines, the **Details** panel holds three tables: Lines, Parents and Children. The **Lines** table displays the line profiling results for the routine selected in the Report panel. This table is empty if the routine was not profiled at line level.

The two other tables are visible regardless of the profiling level. The **Parents** table lists functions and procedures that called the routine selected in the Report panel during the last profiler run. The **Children** table holds routines that the selected routine called. Though the columns in these tables have the same names that the Report columns have, the meaning of these values differ from the meaning of the Report panel values. Click the links below to see what information columns of these tables hold.

- *Parents Table*
- *Children Table*

The **Parents** and **Children** tables may contain the <*JIT compiler*> and <*Garbage collector*> routines. For more information on them, see *<JIT compiler> and <Garbage collector> Routines*.

The columns that display percent values in the Children table (% Time, % with Children, % Misses, % Branches and others) depend on the *Include routine body in Details* item of the Report panel's toolbar. When this item is pressed, AQtime displays a row with the routine body results in the Children table. This changes the number of rows in the table, which, in turn, changes the percent columns' values.

The Details panel displays a chart next to each of the tables. The chart graphically displays information from the table on the left. For instance, it illustrates what child routine or line took more time to execute. To

change the value displayed in the chart, right-click it and select the desired result from the context menu. To customize the chart properties, select Customize Charts from the context menu.

You can arrange the Lines, Parents and Children tables in the Details panel as you wish. To do this, select Allow docking in Details from the context menu and then dock or undock the pages as you would dock or undock other AQtime panels. See *Docking*.

Note that the Details panel does not display all available columns. You can easily add columns to the panel or remove them from it as it is described in *Adding and Removing Columns*.

Parents Table

The **Parents** table is displayed in the Details panel, where you can see the Performance profiler results of your routines. This table lists functions and procedures that called the routine selected in the Report panel during the last profiler run. Though the columns in this table have the same names that the Report columns have, the meaning of these values differ from the meaning of the Report panel values.

Columns of the Parents table are divided into dependent and independent on the active counter. Click the links below to find the description of the desired column.

Note: Some columns of the Parents table display results of the routine selected in the Report panel. In our explanation we will call this routine *the Report panel routine*.

Note on percent columns: The columns that display percent values in the Children table (% Time, % with Children, % Branches, %Misses, % page Faults and others) depend on the *Include routine body in Details* setting that is shown in the toolbar of the Report panel. When this setting is enabled, AQtime displays information on the routine body execution in the Children table (the number of rows in the table changes, which, in turn, changes the percent values).

Columns that do not depend on the active counter

Columns (in alphabetical order)	**Description**
Address	Routine's address in memory. This column is only used for unmanaged (native-code) routines.
Analysis Result	Specifies if the routine was instrumented or not. If the routine was instrumented, this column is empty. Otherwise, the column specifies why the routine was not profiled. These are the same values as the values displayed in the Analysis Result column of the **Report** panel. For more information, see *Performance Profiler - Report Panel Columns*.
Class Name	Name of the class where the parent routine is defined.
Code Type	Specifies the type of the routine's code. The following values are possible: • **MSIL** - Managed-code routine with MSIL (Microsoft Intermediate Language) code. • **x64** - 64-bit code routine. • **x86** - Native-code (unmanaged) routine. • **Pseudo** - Pseudo routine that was created by the context. For example, *<JIT Compiler>*, *<Garbage Collector>*, *<Unknown*

	PInvoke> or <*Root*>. • **PInvoke** - Native-code routine for which there is a declaration in one of managed modules and that is called from within the unmanaged code. • **NGen** - Managed routine that was compiled by the *ngen* utility (CLR Native Image Generator) with the */prof* argument in its command line. The *ngen* compilation means the routine was compiled before the application starts. • **Script** - The routine belongs to a script that was profiled along with the host application. See *Profiling Scripts* for details.
Exceptions	Number of times the parent routine was entered but not successfully exited when calling the Report panel routine. This is usually a count of exceptions that occurred when calling the Report panel routine.
Hit Count	Number of times the parent routine called the Report panel routine.
Module Name	The name of the module which contains the parent routine.
Namespace	Namespace of the method's class (this column is used for managed routines only).
Routine Name	Name of the parent routine.
Source File	Name of the source file for the parent routine. The values for this column are read from the application's debug info.
Source Line	Source files line number where the parent routine's implementation begins. The values for this column are read from the application's debug info.
Token	The routine's token.
Unit Name	Name of the linkage unit that holds the routine. This column is used for unmanaged (native-code) routines only.

Columns that depend on the active counter

***Elapsed Time, User Time* and *User+Kernel Time* Counters**

For each of the following columns you can specify the measurement unit for the displayed values (*seconds*, *milliseconds*, *microseconds* or *machine cycles*) using the *Counter unit* box on the **Profiler** toolbar. See also *Performance Profiler Options*.

Columns (in alphabetical order)	**Description**
Average Time	Average time spent executing the code of the Report panel routine per call. To calculate this value AQtime only uses those calls to the Report panel routine that where made from within the given parent routine.
Average Time with Children	Average time spent executing the Report panel routine per call (including calls to its child routines). To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Max Time **Min Time**	Maximum and minimum time (excluding child function calls) spent executing the Report panel routine when it was called from the parent routine. Exceptional values point out perhaps unexpected special

	conditions.
Max Time with Children **Min Time with Children**	Maximum and minimum time (including child function calls) spent executing the Report panel routine when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Shared Time	Total time spent executing the code of the Report panel routine, as a percentage of the total time spent on calls to the Report panel routine including calls to child routines. In other words, Shared Time is the ratio of the routine's **Time** to **Time with Children** values displayed in the Parents grid: (*Time* / *Time with Children*)**100*
Time	Total time spent executing the Report panel routine's code (excluding child calls) when it was called from the parent routine. Sort results by this column to find the parent routine that uses the Report panel routine more than other parent routines.
Time with Children	Total time spent executing the Report panel routine when it was called from the parent routine. This value includes the time spent executing child routines.
% Time	This is the **Time** value as a percentage of the sum of the Time values displayed in the Parents grid.
% with Children	This is the **Time with Children** value as a percentage of the sum of the Time with Children values displayed in the Parents grid.

Note: The **% Time** and **% with Children** values depend on the *Include routine body in Details* setting that is shown in the toolbar of the Report panel. When this setting is enabled, AQtime displays information on the routine body execution in the Children table. The table contains one more row, and this changes the percent values.

***CPU Cache Misses* Counter**

Columns (in alphabetical order)	**Description**
Average Misses	Average number of CPU cache misses that occurred during execution of the code of the Report panel routine per call. To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Average Misses with Children	Average number of CPU cache misses that occurred during execution of the Report panel routine per call (including cache misses in its child routines). To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Max Misses **Min Misses**	Maximum and minimum number of cache misses that occur during execution of the Report panel routine (child calls are excluded) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Max Misses with Children **Min Misses with Children**	Maximum and minimum number of cache misses that occur during execution of the Report panel routine (including child calls) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.

Misses	Total number of cache misses that occur during execution of the Report panel routine's code (excluding child calls) when it was called from the parent routine.
Misses with Children	Total number of cache misses that occur during execution of the Report panel routine (including its child routines) when it was called from the parent routine.
Shared Misses	Total number of CPU cache misses that occurred during execution of the code of the Report panel routine's code, as a percentage of the total number cache misses that occurred on calls to the Report panel routine (including calls to its child routines). To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine. In other words, Shared Misses is the ratio of the **Misses** to **Misses with Children** values displayed in the Parents grid.
% Misses	This is the **Misses** value as a percentage of the sum of the Misses values displayed in the Parents grid.
% with Children	This is the **Misses with Children** value as a percentage of the sum of the Misses with Children values displayed in the Parents grid.

***CPU Mispredicted Branches* Counter**

Columns (in alphabetical order)	**Description**
Average Branches	Average number of branches that were mispredicted during execution of the own code of the Report panel routine per call. To calculate this value AQtime uses only those calls to the Report panel routine that were made from within the given parent routine.
Average Branches with Children	Average number of branches that were mispredicted during execution of the Report panel routine per call (including mispredictions in its child routines). To calculate this value AQtime uses only those calls to the Report panel routine that were made from within the given parent routine.
Branches	Total number of mispredictions that occurred during execution of the Report panel routine's code (excluding child calls) when it was called from the parent routine.
Branches with Children	Total number of mispredictions that occurred during execution of the Report panel routine (including its child routines) when it was called from the parent routine.
Max Branches **Min Branches**	Maximum and minimum number of mispredictions that occurred during execution of the Report panel routine (child calls are excluded) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Max Branches with Children **Min Branches with Children**	Maximum and minimum number of mispredictions that occurred during execution of the Report panel routine (including child calls) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Shared Branches	Total number of mispredictions that occurred during execution of the Report panel routine's own code, as a percentage of

	mispredictions that occurred during calls to the Report panel routine including calls to its child routines. To calculate this value AQtime uses only those calls to the Report panel routine that were made from within the given parent routine. In other words, this is the ratio of the routine's **Branches** to **Branches with Children** values displayed in the Parents grid.
% Branches	This is the **Branches** value as a percentage of the sum of the Branches values displayed in the Parents grid.
% with Children	This is the **Branches with Children** value as a percentage of the sum of the Branches with Children values displayed in the Parents grid.

***Hard Memory Page Faults, Soft Memory Page Faults* and *All Memory Page Faults* Counters**

Columns (in alphabetical order)	**Description**
Average Faults	Average number of memory page faults that occurred during execution of the own code of the Report panel routine per call. To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Average Faults with Children	Average number of memory page faults that occurred during execution of the Report panel routine per call (including calls to its child routines). To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Faults	Total number of page faults that occurred during execution of the Report panel routine's code (excluding child calls) when it was called from the parent routine.
Faults with Children	Total number of page faults that occurred during execution of the Report panel routine (including its child routines) when it was called from the parent routine.
Max Faults **Min Faults**	Maximum and minimum number of page faults that occurred during execution of the Report panel routine (child calls are excluded) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Max Faults with Children **Min Faults with Children**	Maximum and minimum number of page faults that occurred during execution of the Report panel routine (including child calls) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Shared Faults	Total number of memory page faults that occurred during execution of the Report panel routine's code, as a percentage of the total number of page faults that occurred on execution of the Report panel routine including calls to its child routines. To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine. In other words, this is the ratio of the routine's **Faults** to **Faults with Children** values displayed in the Parents grid.
% Faults	This is the **Faults** value as a percentage of the sum of the Faults values

	displayed in the Parents grid.
% with Children	This is the **Faults with Children** value as a percentage of the sum of the Faults with Children values displayed in the Parents grid.

***Split Load Replays, Split Store Replays and Blocked Store Forwards Replays* Counter**

Columns (in alphabetical order)	**Description**
Average Replays	Average number of replays that occurred during execution of the code of the Report panel routine per call. To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Average Replays with Children	Average number of replays that occurred during execution of the Report panel routine per call (including calls to its child routines). To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Max Replays **Min Replays**	Maximum and minimum number of replays that occurred during execution of the Report panel routine (child calls are excluded) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Max Replays with Children **Min Replays with Children**	Maximum and minimum number of replays that occurred during execution of the Report panel routine (including child calls) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Replays	Total number of replays that occurred during execution of the Report panel routine's code (excluding child calls) when it was called from the parent routine.
Replays with Children	Total number of replays that occurred during execution of the Report panel routine (including its child routines) when it was called from the parent routine.
Shared Replays	Total number of replays that occurred during execution of the Report panel routine's code, as a percentage of the total number of replays that occurred on execution of the Report panel routine including calls to its child routines. To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine. In other words, this is the ratio of the routine's **Replays** to **Replays with Children** values displayed in the Parents grid.
% Replays	This is the **Replays** value as a percentage of the sum of the Replays values displayed in the Parents grid.
% with Children	This is the **Replays with Children** value as a percentage of the sum of the Replays with Children values displayed in the Parents grid.

64K Aliasing Conflicts Counter

Columns (in alphabetical order)	**Description**
Average Conflicts	Average number of aliasing conflicts that occurred during execution of the code of the Report panel routine per call. To

	calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Average Conflicts with Children	Average number of aliasing conflicts that occurred during execution of the Report panel routine per call (including calls to its child routines). To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Conflicts	Total number of aliasing conflicts that occurred during execution of the Report panel routine's code (excluding child calls) when it was called from the parent routine.
Conflicts with Children	Total number of aliasing conflicts that occurred during execution of the Report panel routine (including its child routines) when it was called from the parent routine.
Max Conflicts **Min Conflicts**	Maximum and minimum number of aliasing conflicts that occurred during execution of the Report panel routine (child calls are excluded) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Max Conflicts with Children **Min Conflicts with Children**	Maximum and minimum number of aliasing conflicts that occurred during execution of the Report panel routine (including child calls) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Shared Conflicts	Total number of memory aliasing conflicts that occurred during execution of the Report panel routine's code, as a percentage of the total number of conflicts that occurred on execution of the Report panel routine including calls to its child routines. To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine. In other words, this is the ratio of the routine's **Conflicts** to **Conflicts with Children** values displayed in the Parents grid.
% Conflicts	This is the **Conflicts** value as a percentage of the sum of the Conflicts values displayed in the Parents grid.
% with Children	This is the **Conflicts with Children** value as a percentage of the sum of the Conflicts with Children values displayed in the Parents grid.

***Context Switches* Counter**

Columns (in alphabetical order)	**Description**
Average Switches	Average number of context switches that occurred during execution of the code of the Report panel routine per call. To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Average Switches with Children	Average number of context switches that occurred during execution of the Report panel routine per call (including calls to

	its child routines). To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine.
Max Switches **Min Switches**	Maximum and minimum number of context switches that occurred during execution of the Report panel routine (child calls are excluded) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Max Switches with Children **Min Switches with Children**	Maximum and minimum number of context switches that occurred during execution of the Report panel routine (including child calls) when it was called from the parent routine. Exceptional values point out perhaps unexpected special conditions.
Shared Switches	Total number of context switches faults that occurred during execution of the Report panel routine's code, as a percentage of the total number of switches that occurred on execution of the Report panel routine including calls to its child routines. To calculate this value AQtime only uses those calls to the Report panel routine that were made from within the given parent routine. In other words, this is the ratio of the routine's **Switches** to **Switches with Children** values displayed in the Parents grid.
Switches	Total number of context switches that occurred during execution of the Report panel routine's code (excluding child calls) when it was called from the parent routine.
Switches with Children	Total number of context switches that occurred during execution of the Report panel routine (including its child routines) when it was called from the parent routine.
% Switches	This is the **Switches** value as a percentage of the sum of the Switches values displayed in the Parents grid.
% with Children	This is the **Switches with Children** value as a percentage of the sum of the Switches with Children values displayed in the Parents grid.

Children Table

Columns that do not depend on the active counter

Columns (in alphabetical order)	**Description**
Address	Routine's address in memory. This column is used for unmanaged (native-code) routines only.
Analysis Result	Specifies if the routine was instrumented or not. If the routine was instrumented, this column displays **OK**. Otherwise, the column specifies why the routine was not profiled. These are the same values that are displayed in the Analysis Result column of the Report panel. For more information, see *Performance Profiler - Report Panel Columns*.
Class Name	Name of the class where the child routine is defined.
Exceptions	Number of times the child routine was entered but not successfully exited

	when it was called from the Report panel routine.
Hit Count	Number of times the child routine was called from the Report panel routine.
Module Name	The name of the module which contains the child routine.
Namespace	Namespace of the method's class (this column is used for managed routines only).
Routine Name	Name of the child routine.
Source File	Name of the source file for the child routine. The values for this column are read from the application's debug info.
Source Line	Source files line number where the child routine's implementation begins. The values for this column are read from the application's debug info.
Token	The routine's token.
Unit Name	Name of the linkage unit that holds the routine. This column is used for unmanaged (native-code) routines only.

Columns that depend on the active counter

***Elapsed Time, User Time* and *User+Kernel Time* Counters**

For each of the following columns you can specify the measurement unit for the displayed values (*seconds*, *milliseconds*, *microseconds* or *machine cycles*) using the *Counter unit* box on the **Profiler** toolbar. See also *Performance Profiler Options*.

Columns (in alphabetical order)	**Description**
Average Time	Average time spent executing the child routine's code on one call when the child routine was called from the Report panel routine.
Average Time With Children	Average time spent executing the child routine per on one call when the child routine was called from the Report panel routine.
Max Time **Min Time**	Maximum and minimum time (excluding child function calls) spent executing the child routine when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Max Time with Children **Min Time with Children**	Maximum and minimum time (including child function calls) spent executing the child routine when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Shared Time	Total time spent executing the code of the child routine, as a percentage of the total time spent on calls to the child routine including calls to child routines. In other words, Shared Time is the ratio of the routine's **Time** to **Time with Children** values displayed in the Children grid: (*Time* / *Time with Children*)**100*.
Time	Total time spent executing the child routine's code when it was called from the Report panel routine. This time does not include the execution time of the child functions of this child routine. Sort results by this column to find the slowest child routine among other children of the Report panel routine.
Time with Children	Total time spent executing the child function that was called from the Report panel routine. This value includes the time spent for executing

	child functions of this child routine.
% Time	This is the **Time** value as a percentage of the sum of the Time values displayed in the Children grid.
% with Children	This is the **Time with Children** value as a percentage of the sum of the Time with Children values displayed in the Children grid.

***CPU Cache Misses* Counter**

Columns (in alphabetical order)	**Description**
Average Misses	Average number of CPU cache misses that occurred during execution of the child routine's code per call. To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Average Misses With Children	Average number of CPU cache misses that occurred during execution of the child routine per call (including cache misses in its child routines). To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Max Misses **Min Misses**	Maximum and minimum number of cache misses that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Max Misses with Children **Min Misses with Children**	Maximum and minimum number of cache misses that occurred during execution of the the child routine (including its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Misses	Total number of cache misses that occur during execution of the child routine (excluding its child functions) when it was called from the Report panel routine.
Misses with Children	Total number of cache misses that occur during execution of the child routine (including its child routines) when it was called from the Report panel routine.
Shared Misses	Total number of cache misses that occurred during execution of the code of the child routine, as a percentage of the total number of cache misses that occurred during calls to the child routine including calls to its child routines. To calculate this value AQtime only uses those calls to the child routine that were made from the Report panel routine. In other words, Shared Misses is the ratio of the child routine's **Misses** to **Misses with Children**.
% Misses	This is the **Misses** value as a percentage of the sum of the Misses values displayed in the Children grid.
% with Children	This is the **Misses with Children** value as a percentage of the sum of the Misses with Children values displayed in the Children grid.

***CPU Mispredicted Branches* Counter**

Columns (in alphabetical order)	**Description**

Average Branches	Average number of branches that were mispredicted during execution of the child routine's code per call. To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Average Branches with Children	Average number of branches that were mispredicted during execution of the child routine per call (including mispredictions in its child routines). To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Branches	Total number of mispredictions that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine.
Branches with Children	Total number of mispredictions that occurred during execution of the child routine (including its child routines) when it was called from the Report panel routine.
Max Branches **Min Branches**	Maximum and minimum number of mispredictions that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Max Branches with Children **Min Branches with Children**	Maximum and minimum number of mispredictions that occurred during execution of the the child routine (including its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Shared Branches	Total number of mispredictions that occurred during execution of the code of the child routine, as a percentage of the total number of mispredictions that occurred during execution of the child routine including calls to its child routines. To calculate this value AQtime only uses those calls to the child routine that were made from the Report panel routine. In other words, Shared Branches is the ratio of the child routine's **Branches** to **Branches with Children**.
% Branches	This is the **Branches** value as a percentage of the sum of the Branches values displayed in the Children grid.
% with Children	This is the **Branches with Children** value as a percentage of the sum of the Branches with Children values displayed in the Children grid.

***Hard Memory Page Faults, Soft Memory Page Faults* and *All Memory Page Faults* Counters**

Columns (in alphabetical order)	**Description**
Average Faults	Average number of page faults that occurred during execution of the child routine's code per call. To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Average Faults with Children	Average number of page faults that occurred during execution of the child routine per call (including faults that occurred in its child

	routines). To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Faults	Total number of page faults that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine.
Faults with Children	Total number of page faults that occurred during execution of the child routine (including its child routines) when it was called from the Report panel routine.
Max Faults **Min Faults**	Maximum and minimum number of page faults that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Max Faults with Children **Min Faults with Children**	Maximum and minimum number of page faults that occurred during execution of the the child routine (including its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Shared Faults	Total number of memory page faults that occurred during execution of the code of the child routine, as a percentage of the total number of page faults that occurred during execution of the child routine including calls to its child routines. To calculate this value AQtime only uses those calls to the child routine that were made from the Report panel routine. In other words, Shared Faults is the ratio of the child routine's **Faults** to **Faults with Children**.
% Faults	This is the **Faults** value as a percentage of the sum of the Faults values displayed in the Children grid.
% with Children	This is the **Faults with Children** value as a percentage of the sum of the Faults with Children values displayed in the Children grid.

***Split Load Replays*, *Split Store Replays* and *Blocked Store Forwards Replays* Counter**

Columns (in alphabetical order)	**Description**
Average Replays	Average number of replays that occurred during execution of the child routine's code per call. To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Average Replays with Children	Average number of replays that occurred during execution of the child routine per call (including replays that occurred in its child routines). To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Max Replays **Min Replays**	Maximum and minimum number of replays that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Max Replays with Children **Min Replays with Children**	Maximum and minimum number of replays that occurred during execution of the the child routine (including its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.

Replays	Total number of replays that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine.
Replays with Children	Total number of replays that occurred during execution of the child routine (including its child routines) when it was called from the Report panel routine.
Shared Replays	Total number of replays that occurred during execution of the code of the child routine, as a percentage of the total number of replays that occurred during execution of the child routine including calls to its child routines. To calculate this value AQtime only uses those calls to the child routine that were made from the Report panel routine. In other words, Shared Replays is the ratio of the child routine's **Replays** to **Replays with Children**.
% Replays	This is the **Replays** value as a percentage of the sum of the Replays values displayed in the Children grid.
% with Children	This is the **Replays with Children** value as a percentage of the sum of the Replays with Children values displayed in the Children grid.

***64K Aliasing* Counter**

Columns (in alphabetical order)	**Description**
Average Conflicts	Average number of aliasing conflicts that occurred during execution of the child routine's code per call. To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Average Conflicts with Children	Average number of aliasing conflicts that occurred during execution of the child routine per call (including conflicts that occurred in its child routines). To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Conflicts	Total number of aliasing conflicts that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine.
Conflicts with Children	Total number of aliasing conflicts that occurred during execution of the child routine (including its child routines) when it was called from the Report panel routine.
Max Conflicts **Min Conflicts**	Maximum and minimum number of aliasing conflicts that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Max Conflicts with Children **Min Conflicts with Children**	Maximum and minimum number of aliasing conflicts that occurred during execution of the the child routine (including its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Shared Conflicts	Total number of aliasing conflicts that occurred during execution of the code of the child routine, as a percentage of the total number of conflicts that occurred during execution of the child routine

	including calls to its child routines. To calculate this value AQtime only uses those calls to the child routine that were made from the Report panel routine. In other words, Shared Conflicts is the ratio of the child routine's **Conflicts** to **Conflicts with Children**.
% Conflicts	This is the **Conflicts** value as a percentage of the sum of the Conflicts values displayed in the Children grid.
% with Children	This is the **Conflicts with Children** value as a percentage of the sum of the Conflicts with Children values displayed in the Children grid.

***Context Switches* Counter**

Columns (in alphabetical order)	**Description**
Average Switches	Average number of context switches that occurred during execution of the child routine's code per call. To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Average Switches with Children	Average number of context switches that occurred during execution of the child routine per call (including switches that occurred in its child routines). To calculate this value, AQtime only uses those calls to the child routine that were made from within the Report panel routine.
Max Switches **Min Switches**	Maximum and minimum number of context switches that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Max Switches with Children **Min Switches with Children**	Maximum and minimum number of context switches that occurred during execution of the the child routine (including its child functions) when it was called from the Report panel routine. Exceptional values point out perhaps unexpected special conditions.
Shared Switches	Total number of context switches that occurred during execution of the code of the child routine, as a percentage of the total number of switches that occurred during execution of the child routine including calls to its child routines. To calculate this value AQtime only uses those calls to the child routine that were made from the Report panel routine. In other words, Shared Switches is the ratio of the child routine's **Switches** to **Switches with Children**.
Switches	Total number of context switches that occurred during execution of the child routine (excluding its child functions) when it was called from the Report panel routine.
Switches with Children	Total number of context switches that occurred during execution of the child routine (including its child routines) when it was called from the Report panel routine.
% Switches	This is the **Switches** value as a percentage of the sum of the Switches values displayed in the Children grid.
% with Children	This is the **Switches with Children** value as a percentage of the

	sum of the Switches with Children values displayed in the Children grid.

The **Parents** and **Children** tables may contain the <*JIT compiler*> and <*Garbage collector*> routines. For more information on them, see *<JIT compiler> and <Garbage collector> Routines*.

The **Details** panel displays a chart next to each of the tables. The chart graphically displays information from the table on the left. For instance, it illustrates what child routine or line took more time to execute. To change the value displayed in the chart, right-click it and select the desired result from the context menu. To customize the chart properties, select **Customize Charts** from the context menu.

You can arrange the Lines, Parents and Children tables in the Details panel as you wish. To do this, select **Allow docking in Details** from the context menu and then dock or undock the pages as you would dock or undock other AQtime panels. See *Docking*.

Note that the Details panel does not display all available columns. You can easily add columns to the panel or remove them from it as it is described in *Adding and Removing Columns*.

Performance Profiler Options

The **Performance** profiler includes two groups of customizable options:

- One group includes options that have effect on the profiler functioning. Changes in these options will only apply to the next profiler run.
- Another group contains options that affect the current result displaying. When you change these options, AQtime refreshes data in its panels.

To modify options that affect the profiler functioning, do any of the following:

- *AQtime standalone:*
  - select **Options | Options** from AQtime's main menu and then choose **Profilers | Performance | Performance Profiler** from the tree view on the left of the Options dialog;
  - press **Configure Current Profiler** on the **Standard** toolbar when the **Platform Compliance** profiler is selected.
- *AQtime integrated into Visual Studio:*
  - select **Tools | Options** from Visual Studio's main menu and then choose **AQtime | Profilers | Performance | Performance Profiler** from the tree view on the left of the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - select **Profile | Options** from Borland Developer Studio's main menu and then choose **Profilers | Performance | Performance Profiler** from the tree view on the left of the Options dialog.

Options include:

- **Active counter** - Specifies what application characteristic the profiler will measure. For more information on available values for this option, see *Performance Profiler - Overview*.
- **Thread model** - Specifies how the Performance profiler gathers statistics for threads in the profiled application. For more information on available values for this option, see *Profiling Multiple Threads*.
- **Disable inlining** - Disables method inlining. Inlining typically increases speed and reduces the number of separate JITting events for the inlined methods.

- **Profile <Root> routine** - Specifies if the profiler results include the <*Root*> pseudo-routine. For more information about it, see <*Root*> *Routine*.
- **Profile .NET runtime** - If this option is enabled, the profiler will analyze the JIT compilation and garbage collection and profiling results will include the <*JIT compiler*> and <*Garbage collector*> pseudo-routines. For more information on them, see <*JIT compiler*> *and* <*Garbage collector*> *Routines*. If this option is disabled, the profiler does not monitor calls to the JIT compilation and garbage collection routines.

To modify options that affect the way results are displayed, use items of the **Profiler** toolbar (if AQtime is running as a package within Microsoft Visual Studio, this toolbar is displayed at the top of the Report panel). If this toolbar is hidden, right-click somewhere in the toolbar area and select **Profiler** from the subsequent popup list. The toolbar holds the following items:

- **Counter unit** - This item is enabled only if the Active Counter option is either *Elapsed Time*, *User Time* or *User+Kernel Time*. The *Counter unit* item lets you specify the measurement unit for the time columns in AQtime panels. Possible values are *Seconds*, *Milliseconds*, *Microseconds* and *Machine Cycles*. Note that this option is counter-specific: suppose you browse results of the *User Time* counter and set the option to *Machine Cycles*. If you open the *Elapsed Time* results, change the option to *Seconds* and then return back to the *User Time* results, AQtime will automatically change the option to *Machine Cycles* (that is, it will select the value that was active when you browsed the *User Time* results last time).
- **Show non-hit routines** - Enables or disables the display, in the Report panel, of profiled methods that have not been executed in the current profile run.
- **Calculate "% with children" relative to real values** - If this is enabled, **% with Children** is figured relative to the total **Time** (without children). Otherwise, relative to the total **Time with Children**. See *Calculating Percent in the Report Panel*. Note that this option does not influence the existing profiling results.
- **Include routine body in Details** - Sets whether the results for each routine's own-code ("body") will be listed along with the child-call results in the **Children** table of the **Details** panel.

  This option also affects the values displayed in percent columns (% Time, % with Children, % Misses, % Branches and others). To display routine body results, AQtime adds a new row to the Children panel, which changes the percent columns' values, since they depend on the number of rows.
- **Show routines with class names** - If this option is enabled, the **Routine Name** column of the Report, Details and Call Tree panels for the Performance profiler holds both class name and routine name. Else, this column holds the routine name only.
- **File names with path** - If this option is enabled, the **Source File** and **Module Name** columns of the Report, Details and Call Tree panels for the Performance profiler hold the entire path to the given file. Else, these columns hold the file name only.

Searching for Bottleneck Reasons Using the Performance Profiler

There can be lots of reasons that cause bottlenecks during the application execution. This topic gives you several examples of how to find these causes using the **Performance** profiler. Before reading this topic we recommend you review *Performance Profiler - Overview* and *Analyzing Performance Profiler Results*.

The Performance profiler yields mass statistics about routines in the profiled application: how many times each routines was called, what functions it called and what functions called it, how many exceptions occurred during the routines execution, etc. The profiler includes a number of counters to measure specific application

characteristics. For more information on these counters, see *Performance Profiler - Overview*. The counters help you not just find bottlenecks in your application, but find the cause of these bottlenecks. To be more exact, counters is just one of the means that the Performance profiler provides for finding this cause. Other means include special columns in profiler results, the hierarchy of function calls, profiler options, etc.

Let's continue with an example. Suppose, you profiled your application with the *Elapsed Time* counter and found that FuncA is too slow. The following table gives some examples of how to figure out what caused the bottleneck:

Reason	How to Check
FuncA called several child routines and the bottleneck exists in one of these child routines.	Compare values of the **Time** and **Time with Children** columns in the **Report** panel. If Time With Children is a lot more than Time, then the cause is in one of the child routines. In the **Details** panel, you can easily find how much time each of the child routines contributed to the FuncA execution time.
FuncA called functions from system libraries and the bottleneck exists in one of these functions.	Profile you application with the *User Time* counter and then compare results of two runs.
FuncA worked with memory inefficiently.	Profile your application with the *CPU Cache Misses*, *Soft Memory Page Faults* or *Blocked Store Forwards Replays* counters. High values in profiler results will give evidence that the algorithm for working with memory can be improved.
Most of FuncA's execution time was spent for JIT compilation or garbage collection.	The Performance profiler can time the JIT compilation and garbage collection. To check the time spent for these, view results of the <*JIT compiler*> and <*Garbage collector*> pseudo-routines in the **Details** panel (The Performance profiler includes these routines in results, if the *Profiler .NET runtime* option is enabled).
FuncA contains delayed-initialization code, so most of the time was spent executing the first call to FuncA.	The **Performance** profiler displays profiling results for the first function call separately from the other routine results. To find the cause of the bottleneck, check the **First Time** and **First Time With Children** columns in Report panel (These columns are available if you used the *Elapsed Time*, *User Time* or *User+Kernel Time* counters).

© 2010 AutomatedQA Corp.

Allocation Profiler

Allocation Profiler – Overview

The **Allocation** profiler traces the memory use in 32-bit and 64-bit applications during the application run. It also helps you determine whether allocated memory blocks or objects remain in memory after the application execution is over. This topic provides the Allocation profiler overview. The complete profiler description includes the following topics:

- *Overview (this topic)*
- *Allocation Profiler - Analyzing Visual C++ Applications*
- *Allocation Profiler - Analyzing Visual Basic 6.0 Applications*
- *Allocation Profiler - Analyzing Delphi Applications*
- *Allocation Profiler - Analyzing C++ Builder Applications*
- *Allocation Profiler - Analyzing Intel C++, Borland C++ and GNU CC Applications*
- *Tracing System Memory Management Functions*
- *Checking Bounds of Memory Blocks With the Allocation Profiler*
- *Analyzing Allocation Profiler Results*
- *Description of the Report Panel Columns*
- *Description of the Columns of the Details and Call Tree Panels*
- *Allocation Profiler Options*
- *Using the Monitor Panel With the Allocation Profiler*

Overview

The Allocation profiler tracks the application execution monitoring object allocations and deallocations and calls to the memory management routines. It collects mass statistics, for example:

- The number of objects created in the application
- The size of these objects in memory
- The creation point of an object and the stack of function that calls for it
- References between managed objects
- The usage of allocated memory blocks
- And so forth...

The Allocation profiler traces the memory in both managed and unmanaged (native-code) applications.

Unmanaged (Native-Code) Applications

The Allocation profiler monitors calls to the memory management functions and help determine whether objects and memory blocks remain in memory after the application is over or whether the application writes to memory that does not belong to the allocated block. The profiler traces calls to Windows memory management functions and memory management functions provides by the programming language. The following topics contain detailed information on profiling applications:

- *Analyzing Visual C++ Applications*
- *Analyzing Delphi Applications*
- *Analyzing C++Builder Applications*
- *Analyzing Visual Basic 6.0 Applications*
- *Analyzing Intel C++, Borland C++ and GNU CC Applications*
- *Tracing System Memory Management Functions*

If the *Check Memory Bounds* option is enabled, the profiler traces whether the application wrote to addresses above the upper or below the lower bound of an allocated memory block. For more information, see *Checking Bounds of Memory Blocks*.

By using the profiler you can track memory blocks that are referred to after they have been released. Generally, when a block is released, it is marked as free, but its data may still be available and further instructions may successfully read data from it. If the *Fill released memory blocks* option is enabled, AQtime overwrites the actual data of the block upon its release. So, if there is an attempt to read data from a released block, an invalid value will be returned, which will allow you to find invalid references. See the *Tracing Attempts to Access Released Memory* topic for details.

Managed Applications

When you profile your managed application with the Allocation profiler, AQtime traces all the memory allocated by the application. When the application run is over, the common language runtime reclaims all memory allocated for the application objects. Therefore, after the application has been closed, the memory allocated for it is released. However, Allocation will report that some objects still exist. This happens because the profiler collects final statistics before the run-time calls the garbage collector to free the memory. As the run-time releases memory when the application is being closed, the final statistics may not be very interesting. That is why the Allocation profiler is used mainly to monitor the existing application objects during the run. To obtain results during the application run, select **Run | Get Results** from AQtime's main menu (if AQtime is running as a standalone application), select **Profile | Get Results** from Visual Studio's menu (if AQtime is integrated intoVisual Studio) or click **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar (if AQtime is integrated into Borland Developer Studio).

Another reason why the Allocation profiler can report about unreleased objects after the application termination is that .NET applications can still have actual memory leaks although the Garbage Collector significantly reduces the chance of these leaks. The nature of .NET leaks is different from that of leaks in unmanaged applications. Leaks in unmanaged applications appear if an object or some resource has been allocated but is not released when it is not needed anymore. Leaks in managed applications can appear if root objects (such as global variables) have direct or indirect references to some objects. These objects are not destroyed by the Garbage Collector, and thus their lifetime can equal the application's run-time. Moreover, if objects are not deleted at some points of application execution, you may get memory overflow.

Our Web site holds an article (http://www.automatedqa.com/techpapers/net-allocation-profiler/) that gives examples of potential memory leaks in .NET applications and describes how you can find them using the Allocation profiler of AQtime.

The Allocation profiler traces the use of those objects, which classes are included into profiling areas of the class level (if a class-level area includes a source file, namespace or a module, the profiler analyzes all classes included in that source file, namespace or module). As for memory blocks, the profiler always tracks their allocations and deallocations regardless of any profiling areas.

The Allocation profiler operates during the entire run of the application. It takes no account of triggers and actions and disables the **Enable/Disable Profiling** button.

If you are going to track classes, make sure that you have checked one or more class-level profiling areas before starting profiling. Otherwise, you may get empty results. The reason is quite simple: the profiler *always* tracks memory-block allocations done by non-class memory management routines such as `new` or `alloc`. Therefore, if you start profiling your application and there are no class-level areas selected, the profiler will not notify you, since that application may include calls to non-class memory management routines, which you may want to profile (even .NET applications may include unmanaged sections which hold calls to these routines).

An application may include classes, in which all the methods (including constructors) are inherited, but not overridden (for instance, a class may introduce some new properties or fields, but it may not define new methods or override existing ones). These classes are not listed in the Setup panel, but AQtime will profile them if you enable the *Full Check* option. The Allocation profiler traces instantiation of such classes as allocations of memory blocks, but not as allocations of objects. So, you may notice that the Allocation profiler does not report about leaked objects, but about memory leaks.

To solve the problem, create a new or override an existing method of the class (for instance, the object's constructor).

Note that if a method is not called in your application, the compiler may not include it in the application's binary code. So, even the classes that introduce new methods may become a class described above. To solve the problem in this case, disable optimization in compiler settings.

The Allocation profiler generates a huge amount of results, and sometimes it is difficult to locate the items that are of interest. Therefore AQtime provides a number of built-in filters that hide the results matching certain conditions. Several filters can be applied simultaneously. The buttons that toggle result filtering reside on the **ProfilerReportProfiler** toolbar. The following filters are available:

Show all loaded classes - If enabled, the profiler reports all of the classes being profiled even if no instances (objects) were created for these classes. Otherwise, AQtime only reports the classes whose instances had been created by the time the results were generated.

Filter standard leaks - If enabled, AQtime excludes known memory leaks that were produced by standard IDEs and libraries (like MFC and VCL). Otherwise, these leaks are reported along with the rest of the profiling results. A list of known memory leaks is available at http://www.automatedqa.com/products/aqtime/leaks/.

View project classes only - If enabled, the profiler reports memory allocations and de-allocations that were made only by modules added to the Setup panel. Otherwise it lists the memory operations performed both by the "Setup" modules and by other modules that the "Setup" modules use.

Note: In some applications a class or memory block can be allocated and released by different modules. For instance, a string can be allocated by the main executable and released by a dynamic link library that is used by this executable. If the DLL is not included in your AQtime project, the Allocation profiler will not be able to detect the release of the string and will report a memory leak. This may cause you to think that the main executable has a memory leak, while it does not. To avoid the confusion, include the main executable and the DLLs it uses in your AQtime project.

Filter objects by stack - If enabled, the profiler only reports the objects created immediately by the Setup modules. Otherwise, AQtime reports all created objects for whichever module created it.

The Allocation profiler is closely integrated with the Monitor panel and can display its results during the profiler run, as they are received. See *Using the Monitor Panel With the Allocation Profiler*.

The amount of allocated memory displayed for your application by AQtime may differ from the amount of memory shown in the Task Manager. This happens because AQtime displays the memory that is currently allocated by the application's memory manager for *all live objects being profiled* (the Allocation profiler traces only those objects whose classes are included in profiling areas). In the Task Manager window, you see the memory size that is allocated by the operating system's memory manager for the application. Some part of this memory may not be used at the moment, but it is still allocated by the application's memory manager (for instance, for future use). In certain cases, deallocated memory blocks may not be returned to the operating system's memory manager, so the operating system "thinks" that these blocks are still being allocated by the application. There are also other possible reasons. So, the difference you see is caused by the peculiarities of memory management in the operating system and in the application.

Allocation Profiler – Analyzing Visual C++ Applications

This topic describes the peculiarities of profiling Microsoft Visual C++ applications with the Allocation profiler. The topic includes the following sections:

- *General Information*
- *Preparing Application*
- *Preparing AQtime project*

General Information

The Allocation profiler traces calls to functions that allocate and de-allocate memory blocks and objects. In general, applications can use two function groups to allocate and de-allocate memory: they can use system memory management calls, or they can call on the runtime memory manager, which is part of the runtime library of Visual C++. The runtime memory manager requests large blocks from the system, and then eventually releases them. After that, it works on its own memory-allocation calls from the application. This improves the speed and, most importantly, allows you to avoid system memory thrashing (frequent allocation and de-allocation of small blocks).

The Allocation profiler traces calls to runtime memory manager and system memory management functions. For information on profiling system memory management functions, see *Tracing System Memory Management Functions*. As for runtime memory management functions, the profiler traces the following:

- `new`, `delete`
- `new[]`, `delete[]`
- `malloc`, `calloc`, `realloc`
- `expand`, `free`

In order for AQtime to profile the above-mentioned routines in Visual C++ applications, you may need to add certain modules to the Setup panel in addition to your modules. This depends on the compiler options that were enabled when you compiled your application, specifically enabling the *Debug* configuration. You can find more information on this below. This information is important. If you do not read it, you may get empty results.

The profiler can show the class names when creating and deleting C++ objects. This functionality is supported for 32-bit and 64-bit Visual C++ applications. The profiler traces the objects created dynamically (that is, the object created by the `new` operator). Objects allocated on a stack are not traced (but they do not cause leaks as well).

© 2010 AutomatedQA Corp.

A C++ object will be reported with it's class name if the following condition is met:

- The class must have a constructor written in code or generated by the C++ compiler (compilers typically generate the constructors, if a class is a descendant of another class).

Classes that do not have a constructor are reported under the *C++ native memory* class name (with a complete call stack for each leaked instance). See *Analyzing Allocation Profiler Results* for more information.

Preparing Application

AQtime can track memory usage of Visual C++ applications compiled either in the *Release* or *Debug* configuration. However, applications compiled in the *Release* configuration have certain limitations. Namely:

- If several classes have similar constructors, the linker can exclude the implementation of "redundant" constructors leaving only one of them. As a result, the calls to constructors of different classes will be interpreted as calls to a single constructor, thus the objects of different classes will be treated as the objects of the same class.
- If a class has an empty constructor, the linker excludes the constructor's implementation from compilation and the class is reported under the *C++ native memory* class name in the profiling results.
- The *Release* configuration produces more static allocations of memory. When the application starts, it allocates a block of memory of a predefined size, uses it during the runtime and releases it upon exit. The issue is that the memory is freed after AQtime retrieves the profiling results, so these blocks are reported as leaked.

Therefore, we recommend that you compile your application in the *Debug* configuration. You can do this by specifying the *Debug* compilation mode or by performing the following steps:

- Specify the #define Debug directive in your source code.
- Select **Project | Settings** from the main menu in Visual C++. This will open the **Project Settings** dialog.
- Move to the **C/C++** page.
- Select *General* from the **Category** list and then add _Debug to the preprocessor definitions.
- Select the **Code Generation** category. Choose *Debug Single-Threaded*, *Debug Multithreaded* or *Debug Multithreaded DLL* from the **Use run-time library** dropdown list.
- Press **OK** to close the Project Settings dialog.
- Recompile your application.

Preparing AQtime Project

In order for AQtime to profile the above-mentioned routines in Visual C++ applications, you may need to add certain libraries to the Setup panel in addition to your modules. This depends on the compiler options that were enabled when you compiled your application.

To add a library to the Setup panel, press the **Add Module** button on the Setup toolbar, or select **Add Module** from the panel's context menu. If the required library is not listed in the Modules pane, AQtime will suggest to add it when you start profiling.

To monitor the runtime memory manager, you should add the Microsoft Visual C++ Run Time Library (MSVCRT) to the Modules pane. Depending on whether the application was compiled in the *Release* or *Debug* configuration, either the MSVCRT.DLL or MSVCRTD.DLL (debugging version) file should be added.

By depending on the compiler options, your application can use dynamically linked MFC libraries or statically linked MFC libraries. AQtime can track memory manager functions regardless of which MFC version you use - dynamic or static. However, when AQtime tracks the call stack of memory management functions, it will track only those routines that are shown in the Modules pane. So, if you want to see a more detailed call stack, you need to add MFC libraries to the Setup panel.

Let's continue with a simple example. Suppose, your application contains the fooA function that calls MFC's function CDC::GetPen, which, in turn, calls the malloc routine located in the MSVCRTD dynamic link library. The malloc routine, in its turn, can call debug_malloc or some other functions. The hierarchy of function calls looks like this:

```
fooA (your exe) -> CDC::GetPen (mfc71d.dll) -> malloc (msvcrtd.dll)
                   -> debug_malloc (msvcrtd.dll)
```

If you do not add MFC71 and MSVCRTD libraries to the Setup panel, the call stack will hold only the fooA function. If you add these libraries to the Setup panel, you will get a more detailed call stack.

Allocation Profiler – Analyzing Visual Basic 6.0 Applications

This topic describes peculiarities of analyzing Visual Basic 6.0 applications with the Allocation profiler. Profiling of Visual Basic .NET applications does not differ from profiling of other .NET applications. For more information on this, see the description of the Allocation profiler.

The Allocation profiler tracks functions that allocate or de-allocate memory. In general, applications can do this in two ways: they can use system memory management calls, or they can call the runtime memory manager that is part of Visual Basic's runtime library. The runtime memory manager requests large blocks from the system, and then releases them when it is needed. It then deals on its own with the application's memory-allocation calls. This improves functioning speed, and most importantly, allows you to avoid system memory thrashing (frequent allocations and de-allocations of small blocks).

The Allocation profiler traces calls both to VB memory manager's functions and memory management functions provided by Windows API. For information on profiling system memory management functions, see *Tracing System Memory Management Functions*. As for runtime memory management functions, the profiler traces calls to the following:

- `__vbaRedim`, `__vbaFreeObj`, `__vbaStrCopy`, `__vbaAryDestruct`
- `__vbaFreeStr`, `__vbaNew`, `__vbaStrMove`

These functions are called upon creation and reallocation of arrays, upon creation of classes that refer to COM objects, upon creation of classes with string fields, and so forth.

Creation and deletion of VB classes is traced as creation and deletion of memory blocks. That is, the profiler results do not include the names of leaked VB classes; the leaked classes (if any) are reported under the *VB native memory* class name (with a complete call stack for each leaked instance). See also *Analyzing Allocation Profiler Results*.

To profile Visual Basic 6.0 applications with the Allocation profiler, add the MSVBVM60.DLL library to your AQtime project (see *Opening a Project* to learn how to do this). By default, the DLL is in the <Windows>\System32 folder. If the DLL is not added to the project, AQtime will not start profiling.

© 2010 AutomatedQA Corp.

The Visual Basic memory manager allocates memory when a new object is created in your application. When the application is closed, the memory manager automatically releases all memory blocks it allocated during the application run. Thus, the Allocation profiler will not report any memory leaks in your Visual Basic application. However, this profiler, along with the Monitor panel, can be helpful if you want to explore how your application uses memory in real time. The profiler reports a call stack for each call to memory manager routines, so you can easily see calls to which functions and procedures increase the amount of memory used by your application. The call stack does not include rows for memory manager's routines (these routines are internal routines of Visual Basic's memory manager, so there is no need to know them, since they do not contain useful information).

Note that since the memory manager allocates large memory blocks and then distributes them according to memory allocation calls from the application, memory allocations made through the memory manager functions may not increase the total amount of memory allocated for the application.

Allocation Profiler – Analyzing Delphi Applications

This topic describes peculiarities of profiling Delphi applications with the **Allocation** profiler. The topic includes the following sections:

- *About Routine and Class Profiling*
- *Preparing AQtime Project*
- *Empty Line in the Call Stack*
- *Duplicated memory leaks*

Note: In order for AQtime to be able to profile your Delphi application, please set compiler options as it is described in the *Compiler Settings for Borland Delphi* topic.

About Routine and Class Profiling

The Allocation profiler tracks functions that allocate or de-allocate memory. In general, applications can do this in two ways: they can use system memory management calls, or they can call the runtime memory manager, which is part of the VCL runtime library. The runtime memory manager requests large blocks from the system, and then eventually releases them. After that, it works on its own with a lot of memory-allocation calls from the application. This improves speed, and more importantly, allows you to avoid system memory thrashing (frequent allocation and de-allocation of small blocks).

The Allocation profiler traces calls to runtime memory manager and system memory management functions. For information on profiling system memory management functions, see *Tracing System Memory Management Functions*. The profiler traces calls to the following runtime memory management functions:

- `GetMem`, `ReallocMem`, `FreeMem`
- `GetMemory`, `ReallocMemory`, `FreeMemory`
- `SysGetMem`, `SysReallocMem`, `SysFreeMem`

The `New` and `Dispose` routines are not traced explicitly. They call `GetMem` and the profiler traces these calls.

The profiler is able to show the class names when creating and deleting VCL objects. Note, that an object will be reported with its class name if several conditions are met:

- The class is inherited from TObject.
- The class introduces new methods or override the inherited TObject methods.

- You call those methods from your code. (Otherwise they can be excluded from the debug information by the compiler.)

In any other case, for example, if class only introduces properties and fields, the leaked classes (if any) are reported under the *VCL native memory* class name (with a complete call stack for each leaked instance). See also *Analyzing Allocation Profiler Results*.

In order for AQtime to profile the above-mentioned routines in Delphi applications, you may need to add certain modules to the Setup panel in addition to your modules. This depends on compiler options that were enabled when you compiled your application, specifically enabling/disabling the *Build with runtime packages* option. More information on this is below. Please read it, as this information is important. If you do not read it, you may get empty results.

Preparing AQtime project

The Allocation profiler tracks both calls to runtime memory manager and system memory management functions (see above). This section describes peculiarities of profiling runtime memory manager functions.

The memory manager can be located in the profiled module or in one of the packages. This depends on the ***"Build with runtime packages"*** compiler option (you can change it on the **Packages** tabbed page of the **Project Options** dialog):

- If this option is turned on, the memory manager code is not included in the executable and the application uses the memory manager from the .bpl runtime package located in the <Windows>\System32 folder. For instance, vcl50.bpl for Delphi 5, rtl60.bpl for Delphi 6 or rtl70.bpl for Delphi 7. If your application was compiled with the *"Build with runtime packages"* option enabled and you want to profile calls to the memory manager, you should add this package to the **Setup** panel. The package can be compiled without debug information.

  At the beginning of the Allocation profiler, AQtime checks whether memory management routines are contained within packages. If the check is successful, it shows a message informing you which packages should be added to your AQtime project.
- If the "Build with runtime packages" option is off, the memory manager is located within the profiled executable. In this case, there is no need to add bpl modules to the Setup panel.

Empty lines in the Call Stack

When AQtime tracks the hierarchy of function calls that led to an object creation or allocation of a memory block, it traces only those routines for which it can find debug information. When you create a new class instance (that is, an object), a memory management routine is not called directly by the class constructor. A call to it can be made by other routines which the class constructor calls. These routines typically locate in VCL units. Since these units may be compiled without debug information, the call stack may not hold all the routines, which were called, or it may hold partial information for them (for instance, information about source lines can be absent). Therefore, to obtain more detailed call stack, compile your application with the *Use Debug DCUs* option enabled. You can change the option on the **Linker** tabbed page of the Project Options dialog.

Duplicated memory leaks

When profiling a VCL application, the Allocation profiler results may include duplicated memory leaks. Why does this happen? Creation of an object implies allocation of a memory block for that object. The Allocation profiler tracks both object creation and memory block allocation. Therefore, the profiling results include both object leaks and memory block leaks. Suppose that your application creates some objects and does not delete them after termination. In this case, profiling results will include duplicated memory leaks (there will be object leaks and memory block leaks that concern the same memory allocation). To avoid such

duplication, enable the *Remove duplicate allocations* profiler option. When this option is on, the Allocation profiler adds only object leaks to profiling results, duplicated memory block leaks are hidden.

Allocation Profiler – Analyzing C++Builder Applications

This topic describes peculiarities of profiling C++Builder applications with the **Allocation** profiler. The topic includes the following sections:

- *About Routine and Class Profiling*
- *Preparing AQtime Project*
- *Empty Line in the Call Stack*
- *Duplicated memory leaks*

Note: In order for AQtime to be able to profile your C++Builder application, please set compiler options as it is described in the *Compiler Settings for Borland C++ Builder* topic.

About Routine and Class Profiling

The Allocation profiler tracks functions that allocate or de-allocate memory. In general, applications can do this in two ways: they can use system memory management calls, or they can call the runtime memory manager, which is part of the VCL runtime library. The runtime memory manager requests large blocks from the system, and then eventually releases them. After that, it works on its own with a lot of memory-allocation calls from the application. This improves the speed, and more importantly, allows you to avoid system memory thrashing (frequent allocation and de-allocation of small blocks).

The Allocation profiler traces calls runtime memory manager and system memory management functions. For information on profiling system memory management functions, see *Tracing System Memory Management Functions*. The profiler traces calls to the following runtime memory management functions:

- **VCL**
  - `GetMem`, `ReallocMem`, `FreeMem`
  - `GetMemory`, `ReallocMemory`, `FreeMemory`
  - `SysGetMem`, `SysReallocMem`, `SysFreeMem`

  The `New` and `Dispose` routines are not traced explicitly. They call `GetMem` and the profiler traces these calls.

  To enable AQtime to profile the above-mentioned routines in C++Builder applications, you may need to add certain modules to the Setup panel in addition to your modules. This depends on compiler options that were enabled when you compiled your application, specifically enabling/disabling the *Build with runtime packages* option. More information on this is below. Please read it, as this information is important. If you do not read it, you may get empty results.

- **C++**
  - `new, delete`
  - `new[], delete[]`
  - `malloc, calloc, realloc`
  - `expand, free`

The profiler can also show class names reporting object leaks. Note, that an object will be reported with it's class name if the following conditions are met:

- **VCL**
  - `The class must be inherited from TObject.`
  - `The class must introduce new methods or override methods derived from TObject.`
  - `The class methods should be called in your code (otherwise they can be excluded from the debug information by the compiler.)`

  In any other case, for example, if class only introduces properties and fields, the leaked classes (if any) are reported under the VCL native memory class name (with a complete call stack for each leaked instance). See also *Analyzing Allocation Profiler Results*.
- **C++**
  - `The class must have a constructor written in code or generated by the C++ compiler (compilers typically generate the constructors, if a class is a descendant of another class).`

  Classes that do not have a constructor are reported under the *C++ native memory* class name (with a complete call stack for each leaked instance). See *Analyzing Allocation Profiler Results* for more information.

Preparing AQtime project

The **Allocation** profiler tracks functions that allocate or de-allocate memory. The functions can do this in two ways: they can use system memory management calls, or they can call on the runtime memory manager, which is part of the VCL runtime library. The runtime memory manager requests large blocks from the system, and then eventually releases them. After that, it works on its own with a lot of memory-allocation calls from the application. This improves the speed and, which is more important, allows you to avoid system memory thrashing (frequent allocation and de-allocation of small blocks).

The memory manager can be located in the profiled module or in one of the packages. This depends on the ***"Build with runtime packages"*** compiler option (you can change it on the **Packages** tabbed page of the **Project Options** dialog):

- If this option is turned on, the memory manager code is not included in the executable and the application uses the memory manager from the .bpl runtime package located in the <Windows>\System32 folder, for instance, rtl60.bpl for C++Builder 6. If your application was compiled with the *"Build with runtime packages"* option enabled and you want to profile calls to the memory manager, you should add this package to the Setup panel. The package can be compiled without debug information.

  At start of the Allocation profiler AQtime checks whether memory management routines are contained within packages. If the check is successful, it shows a message informing you which packages should be added to your AQtime project.
- If the *"Build with runtime packages"* option is off, the memory manager is located within the profiled executable. In this case, there is no need to add bpl modules to the Setup panel.

Note: In order for AQtime to be able to profile VCL code, please set compiler options as it is described in the *Compiler Settings for Borland C++Builder* topic.

Empty lines in the Call Stack

When AQtime tracks the hierarchy of function calls that led to an object creation or allocation of a memory block, it traces only those routines for which it can find debug information. When you create a new class instance (i.e. an object), a memory management routine is not called directly by the class constructor. A call to it can be made by other routines which the class constructor calls. These routines typically locate in VCL units. Since these units may be compiled without debug information, the call stack may not hold all the routines, which were called, or it may hold partial information for them (for instance, information about source lines can be absent). Therefore, to obtain more detailed call stack, compile your application with the *Use Debug DCUs* option enabled (in C++Builder the option name is *Use debug libraries*). You can change the option on the **Linker** tabbed page of the Project Options dialog.

Duplicated memory leaks

When profiling a VCL application, the Allocation profiler results may include duplicated memory leaks. Why does this happen? Creation of an object implies allocation of a memory block for that object. The Allocation profiler tracks both object creation and memory block allocation. Therefore, the profiling results include both object leaks and memory block leaks. Suppose that your application creates some objects and does not delete them after termination. In this case, profiling results will include duplicated memory leaks (there will be object leaks and memory block leaks that concern the same memory allocation). To avoid such duplication, enable the *Remove duplicate allocations* profiler option. When this option is on, the Allocation profiler adds only object leaks to profiling results, duplicated memory block leaks are hidden.

Allocation Profiler – Analyzing Intel C++, Borland C++ and GNU CC Applications

The Allocation profiler traces calls to functions that allocate and de-allocate memory blocks and objects. In general, applications can use two function groups to allocate and de-allocate memory: they can use system memory management calls, or they can call on the runtime memory manager, which is part of the runtime library of Intel C++, Borland C++ and GNU CC. The runtime memory manager requests large blocks from the system, and then eventually releases them. After that, it works on its own with a lot of memory-allocation calls from the application. This improves speed, and most importantly, allows you to avoid system memory thrashing (frequent allocation and de-allocation of small blocks).

The Allocation profiler traces calls to the C++ memory manager's functions and memory management functions provided by Windows API. For information on profiling system memory management functions, see *Tracing System Memory Management Functions*. The profiler traces calls to the following runtime memory management functions:

- `new`, `delete`
- `new[]`, `delete[]`
- `malloc`, `calloc`, `realloc`
- `expand`, `free`

To enable AQtime to profile the above-mentioned routines in your Intel C++, Borland C++ or GNU CC application, you may need to add certain modules to the Setup panel in addition to your modules. This depends on the compiler options that were enabled when you compiled your application, specifically enabling the *Debug* configuration.

Creation and deletion of C++ classes located in Intel C++, Borland C++ and GNU CC applications are traced as creation and deletion of memory blocks. That is, the profiler results do not include the names of leaked C++ classes; the leaked classes (if any) are reported under the *C++ native memory* class name (with a complete call stack for each leaked instance). See also *Analyzing Allocation Profiler Results*.

Tracing System Management Memory Functions

The **Allocation** profiler tracks functions that allocate or de-allocate memory. Applications work with memory using functions of two types: system memory management functions (Windows API functions) or functions of the runtime memory manager, which is part of the VCL runtime library. For information on profiling runtime memory manager functions, see the topics listed in the See Also section.

The profiler always traces calls to functions of the runtime memory manager. As for calls to system memory management functions, they are traced only if the profiler's *Check system memory allocations* option is enabled. By default, this option is disabled. If you enable it, the profiler will trace calls to the following Windows API functions that allocate and de-allocate memory:

CommandLineToArgvW	LocalFree
GlobalAlloc	LocalReAlloc
GlobalFree	FormatMessage
GlobalReAlloc	PrintDlgA
HeapAlloc	PrintDlgW
HeapCreate	ReleaseStgMedium
HeapDestroy	SetClipboardData
HeapFree	VirtualAlloc
HeapReAlloc	VirtualFree
LocalAlloc	

Memory blocks that were allocated by system memory management functions and not disposed at the end of the application run are reported under specific class names (with a complete call stack for each leaked block). The following table displays the relationship between functions and "leaks' class names":

"Class Name"	Functions
Committed Virtual Memory, Reserved Virtual Memory	VirtualAlloc, VirtualFree
Global heap	GlobalAlloc, GlobalFree, GlobalReAlloc, PrintDlgA, PrintDlgW, ReleaseStgMedium, SetClipboardData
Heap	HeapAlloc, HeapCreate, HeapDestroy, HeapFree, HeapReAlloc
Heap memory	VirtualAlloc, VirtualFree
Local heap	CommandLineToArgvW, FormatMessage, LocalAlloc, LocalFree, LocalReAlloc

Tracing Attempts to Access Released Memory

Many applications release memory, but attempt to access it later. As a result, the application may crash, raise an exception, cause an access violation error, etc. This situation is known as **use after free** or **premature free**, and the remaining reference to the memory is known as a **dangling pointer**. The situation usually occurs when one part of the application "decides" that it has finished using a memory block and is unaware that another part of the application is still using it.

Sometimes, however, it may be difficult to reveal premature free because of the following peculiarity: When a memory block is deallocated, it is marked as free in a special list, but the actual data remains in memory unless the block is overwritten by other instructions. So, attempts to read data referenced by a dangling pointer may still be successful.

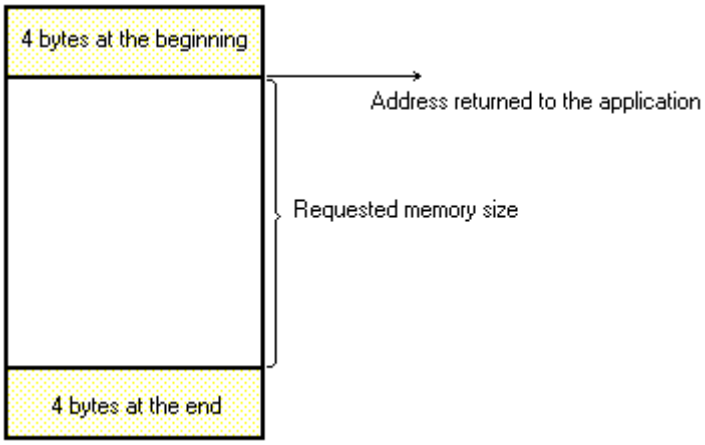
To track the cases of premature free, you can use the Allocation profiler. When the profiler's *Fill released memory blocks* option is enabled, AQtime overwrites the block contents with the 0xDD values. As a result, the application does not read valid data from freed blocks, so, any attempts to read such data will result in errors and will be reported in the Event View panel.

For each error, the Event View panel logs the exception code and description, as well as the call stack that led to the exception. In the call stack, the topmost routine is the one where the exception occurred. If the application was compiled with debug information, then the Editor panel can display the source code for the routine selected in the call stack. Debug info also lets you distinguish routines of the call stack easier. Without it, only the addresses of these routines are displayed. In this case, names are available only for functions that are exported from DLLs. In debug info, routine names are used in their natural format.

Checking Bounds of Memory Blocks

The **Allocation** profiler includes an option, *Check memory bounds*, which specifies whether the profiler reports an error when the profiled application writes to addresses above the upper or below the lower bound of an allocated memory block.

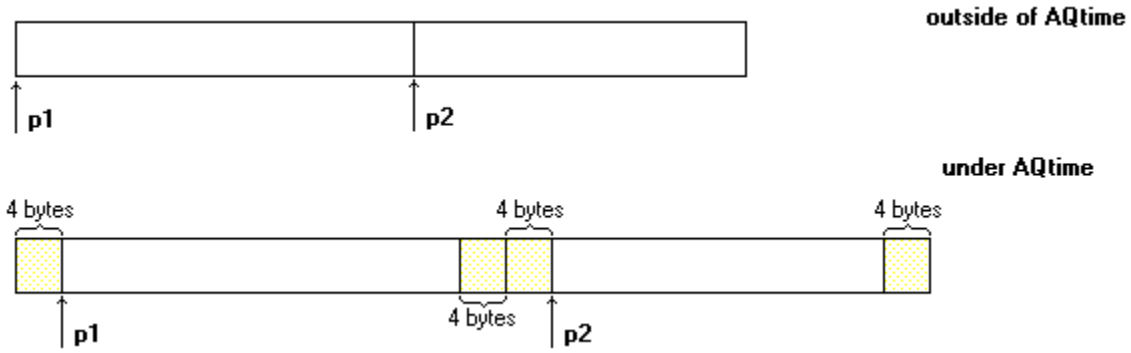
To implement this check, the profiler hooks functions that allocate memory blocks. It returns a block allocated for 8 bytes more than requested, but the application is only informed of owning the size it requested. 4 bytes are reserved before the requested block, and 4 are reserved after --



Important notes:

- The profiler leaves its own signatures in each 4-byte buffer. If the application overwrites the memory outside of these 4-byte buffers, AQtime will not report an error.
-
- AQtime determines that the bounds were exceeded when it finds a 4-byte signature that was overwritten. It checks for it in the following situations:
  - When the block is re-allocated.
  - When the block is released.
  - When the application is terminated.
  - When the results are generated via the **Get Results** command (see *Getting Profiling Results During Testing*).
- The application may be coded on assumptions concerning memory, which should not be made, but "generally work". Therefore it may work well outside of AQtime and misbehave under AQtime

with the *Check memory bounds* option enabled. For instance, it might allocate two consecutive blocks of memory, then attempt to fill them in one call to ZeroMemory():



This ruins the memory-bounds checking, but it also constitutes unexpected conditions for the application itself. In general, applications should not be coded on the assumption that they control the order of allocations from the memory manager. The workaround is to disable *Check memory bounds*. If the problem goes away, then this was probably the cause.

- Due to certain peculiarities of AQtime, the Clear Results command does not work if the *Check memory bounds* setting is enabled.
- Since the bounds-checking control causes additional memory spaces to be allocated, some tools that trace the memory usage (for example, Task Manager) will slightly exaggerate the memory used by your application.
- The *Check memory bounds* feature only operates with native-code applications. It does not work with managed code applications.
- AQtime does not track allocations done on the stack, therefore you cannot track violations in the stack memory blocks.

The Allocation profiler displays the bounds check results along with information about existing classes and instances. The **Classes** category of the Report panel contains the *Memory Overwrite Error* row. The presence of this row indicates that the profiler detected a violation of memory block bounds.

The **Objects** category contains a row for every memory violation, which the profiler detected. The row contains the text *Memory Overwrite Error.nn* (where nn specifies the number of the detected violations).

The Details panel displays the stack of function calls corresponding to the row that is selected in the Report panel. Note that the profiler detects the memory corruption in one of the following situations: when a memory block is released, when the block is reallocated, when the application is terminated or when the Get Results command is executed (see above). The call stack that is shown in the Details panel corresponds to the situation and if it is impossible to trace the call stack, it will be empty:

- If the profiler detects a memory corruption when a block is deleted of reallocated, the call stack will display the sequence of function calls that led to the block deletion or reallocation. The call stack will not point to the routine that includes the code statements that violated memory block bounds.
- If a corrupted block is detected when the application terminates, the profiler is unable to trace the call stack so the call stack appears empty.

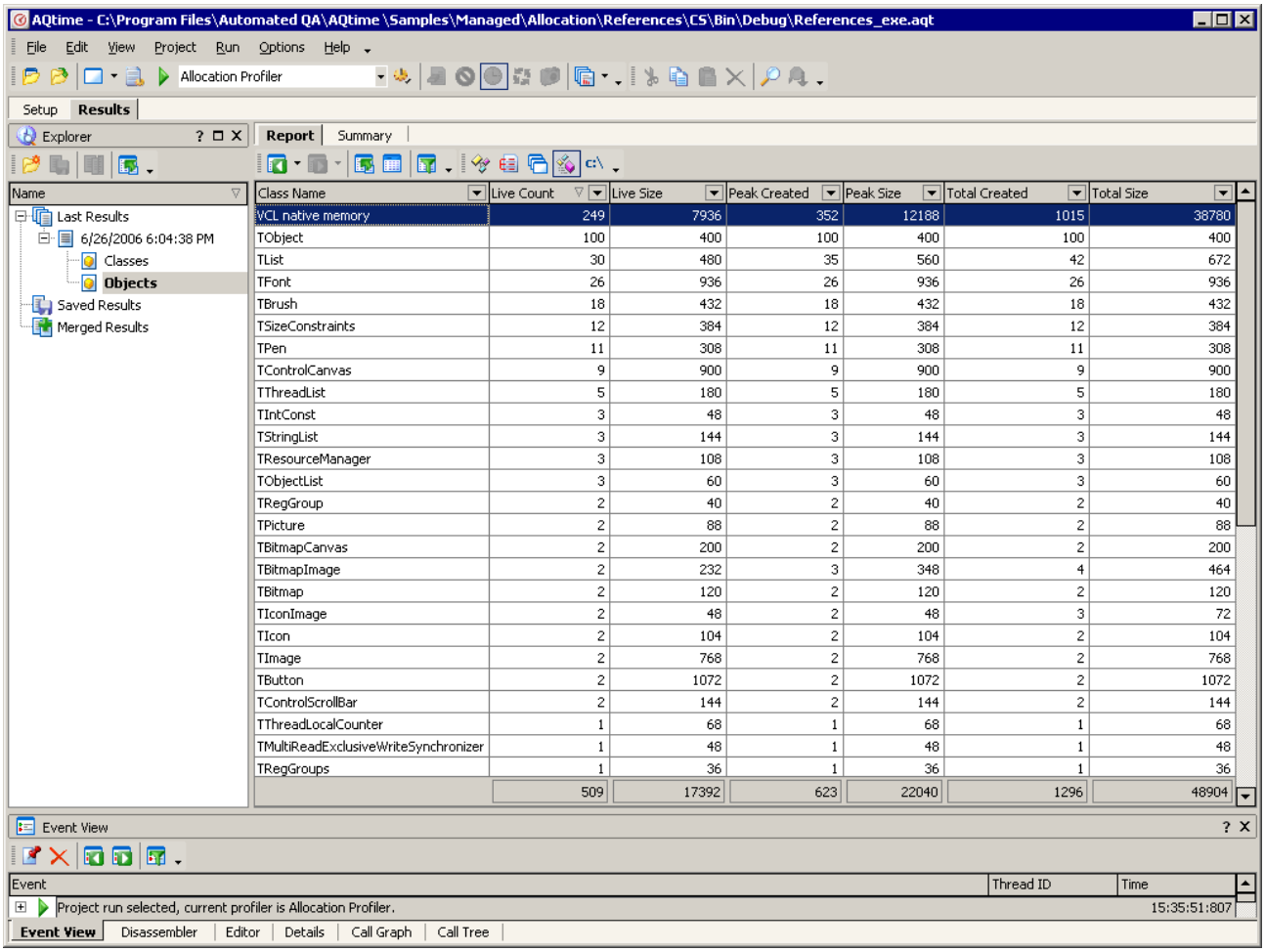
  The same happens if the memory corruption is detected when the Get Results command is working since in this case there is no routine for which the call stack information can be gathered.

© 2010 AutomatedQA Corp.

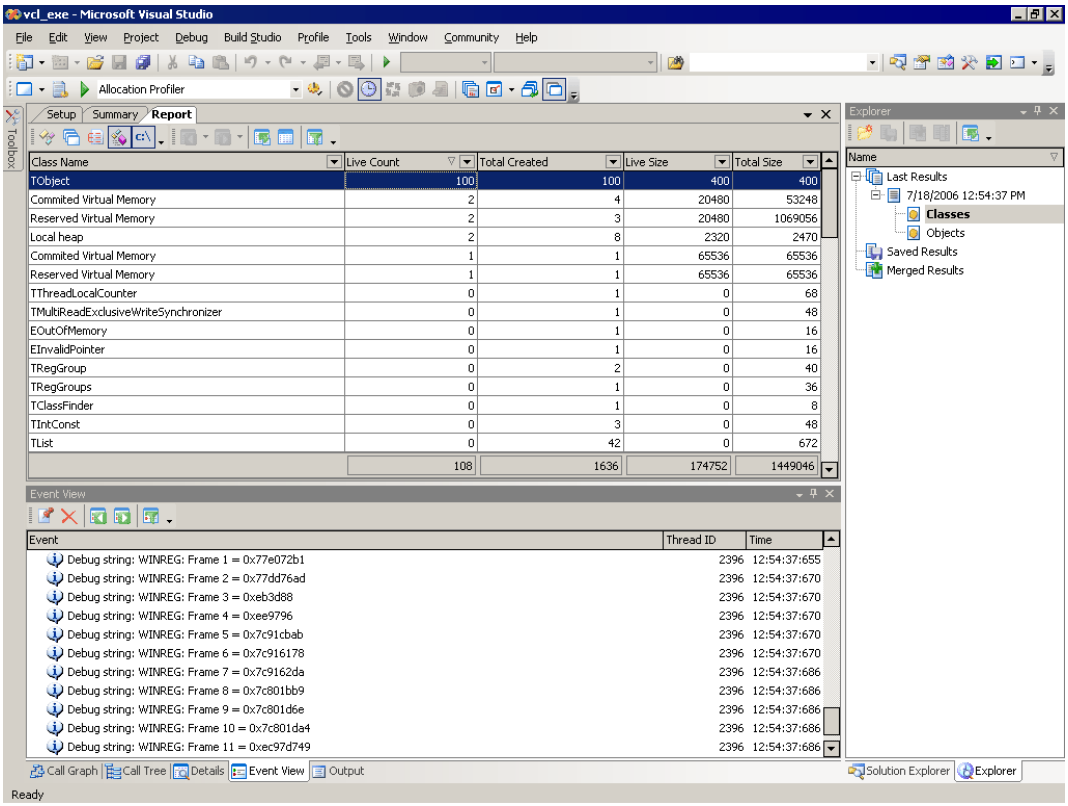
Analyzing Allocation Profiler Results

The **Allocation** profiler traces the use of classes that are added to class-level profiling areas. Like other AQtime profilers, Allocation generates results upon selecting **Run | Get Results** from AQtime's main menu, **Profile | Get Results** from Visual Studio's menu, clicking **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar, or after the application terminates.

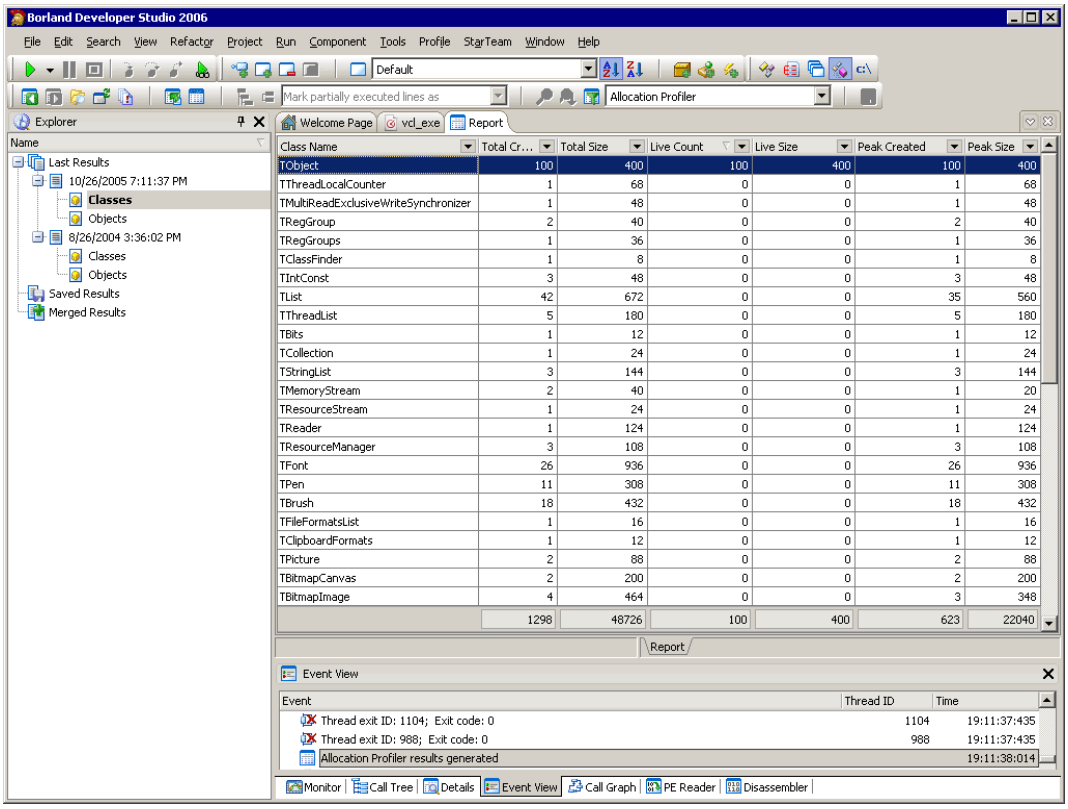
The **Summary** panel displays brief profiling results. It reports about classes with maximum number of existing instances, classes with maximum number of created instances and so on. Information about about all the classes and their instances that are used by the application is shown in the **Report** panel. Here is a sample output of the Allocation profiler:


AQtime - C:\Program Files\Automated QA\AQtime \Samples\Managed\Allocation\References\CS\Bin\Debug\References_exe.aqt
File Edit View Project Run Options Help
Allocation Profiler
Setup Results
Explorer
Report Summary
Name
Last Results
6/26/2006 6:04:38 PM
Classes
Objects
Saved Results
Merged Results
Class Name | Live Count | Live Size | Peak Created | Peak Size | Total Created | Total Size
VCL native memory 249 7936 352 12188 1015 38780
TObject 100 400 100 400 100 400
TList 30 480 35 560 42 672
TFont 26 936 26 936 26 936
TBrush 18 432 18 432 18 432
TSizeConstraints 12 384 12 384 12 384
TPen 11 308 11 308 11 308
TControlCanvas 9 900 9 900 9 900
TThreadList 5 180 5 180 5 180
TIntConst 3 48 3 48 3 48
TStringList 3 144 3 144 3 144
TResourceManager 3 108 3 108 3 108
TObjectList 3 60 3 60 3 60
TRegGroup 2 40 2 40 2 40
TPicture 2 88 2 88 2 88
TBitmapCanvas 2 200 2 200 2 200
TBitmapImage 2 232 3 348 4 464
TBitmap 2 120 2 120 2 120
TIconImage 2 48 2 48 3 72
TIcon 2 104 2 104 2 104
TImage 2 768 2 768 2 768
TButton 2 1072 2 1072 2 1072
TControlScrollBar 2 144 2 144 2 144
TThreadLocalCounter 1 68 1 68 1 68
TMultiReadExclusiveWriteSynchronizer 1 48 1 48 1 48
TRegGroups 1 36 1 36 1 36
509 17392 623 22040 1296 48904
Event View
Event Thread ID Time
Project run selected, current profiler is Allocation Profiler. 15:35:51:807
Event View Disassembler Editor Details Call Graph Call Tree

AQtime standalone


vcl_exe - Microsoft Visual Studio
File Edit View Project Debug Build Studio Profile Tools Window Community Help
Allocation Profiler
Setup Summary Report
Class Name | Live Count | Total Created | Live Size | Total Size
TObject | 100 | 100 | 400 | 400
Commited Virtual Memory | 2 | 4 | 20480 | 53248
Reserved Virtual Memory | 2 | 3 | 20480 | 1069056
Local heap | 2 | 8 | 2320 | 2470
Commited Virtual Memory | 1 | 1 | 65536 | 65536
Reserved Virtual Memory | 1 | 1 | 65536 | 65536
TThreadLocalCounter | 0 | 1 | 0 | 68
TMultiReadExclusiveWriteSynchronizer | 0 | 1 | 0 | 48
EOutOfMemory | 0 | 1 | 0 | 16
EInvalidPointer | 0 | 1 | 0 | 16
TRegGroup | 0 | 2 | 0 | 40
TRegGroups | 0 | 1 | 0 | 36
TClassFinder | 0 | 1 | 0 | 8
TIntConst | 0 | 3 | 0 | 48
TList | 0 | 42 | 0 | 672
108 1636 174752 1449046
Explorer
Name
Last Results
7/18/2006 12:54:37 PM
Classes
Objects
Saved Results
Merged Results
Event View
Event | Thread ID | Time
Debug string: WINREG: Frame 1 = 0x77e072b1 2396 12:54:37:655
Debug string: WINREG: Frame 2 = 0x77dd76ad 2396 12:54:37:670
Debug string: WINREG: Frame 3 = 0xeb3d88 2396 12:54:37:670
Debug string: WINREG: Frame 4 = 0xee9796 2396 12:54:37:670
Debug string: WINREG: Frame 5 = 0x7c91cbab 2396 12:54:37:670
Debug string: WINREG: Frame 6 = 0x7c916178 2396 12:54:37:670
Debug string: WINREG: Frame 7 = 0x7c9162da 2396 12:54:37:686
Debug string: WINREG: Frame 8 = 0x7c801bb9 2396 12:54:37:686
Debug string: WINREG: Frame 9 = 0x7c801d6e 2396 12:54:37:686
Debug string: WINREG: Frame 10 = 0x7c801da4 2396 12:54:37:686
Debug string: WINREG: Frame 11 = 0xec97d749 2396 12:54:37:686
Call Graph Call Tree Details Event View Output
Solution Explorer Explorer
Ready

AQtime integrated into Visual Studio


Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Default
Mark partially executed lines as
Allocation Profiler
Explorer
Name
Last Results
10/26/2005 7:11:37 PM
Classes
Objects
8/26/2004 3:36:02 PM
Classes
Objects
Saved Results
Merged Results
Welcome Page vcl_exe Report
Class Name | Total Cr... | Total Size | Live Count | Live Size | Peak Created | Peak Size
TObject | 100 | 400 | 100 | 400 | 100 | 400
TThreadLocalCounter | 1 | 68 | 0 | 0 | 1 | 68
TMultiReadExclusiveWriteSynchronizer | 1 | 48 | 0 | 0 | 1 | 48
TRegGroup | 2 | 40 | 0 | 0 | 2 | 40
TRegGroups | 1 | 36 | 0 | 0 | 1 | 36
TClassFinder | 1 | 8 | 0 | 0 | 1 | 8
TIntConst | 3 | 48 | 0 | 0 | 3 | 48
TList | 42 | 672 | 0 | 0 | 35 | 560
TThreadList | 5 | 180 | 0 | 0 | 5 | 180
TBits | 1 | 12 | 0 | 0 | 1 | 12
TCollection | 1 | 24 | 0 | 0 | 1 | 24
TStringList | 3 | 144 | 0 | 0 | 3 | 144
TMemoryStream | 2 | 40 | 0 | 0 | 1 | 20
TResourceStream | 1 | 24 | 0 | 0 | 1 | 24
TReader | 1 | 124 | 0 | 0 | 1 | 124
TResourceManager | 3 | 108 | 0 | 0 | 3 | 108
TFont | 26 | 936 | 0 | 0 | 26 | 936
TPen | 11 | 308 | 0 | 0 | 11 | 308
TBrush | 18 | 432 | 0 | 0 | 18 | 432
TFileFormatsList | 1 | 16 | 0 | 0 | 1 | 16
TClipboardFormats | 1 | 12 | 0 | 0 | 1 | 12
TPicture | 2 | 88 | 0 | 0 | 2 | 88
TBitmapCanvas | 2 | 200 | 0 | 0 | 2 | 200
TBitmapImage | 4 | 464 | 0 | 0 | 3 | 348
1298 48726 100 400 623 22040
Report
Event View
Event | Thread ID | Time
Thread exit ID: 1104; Exit code: 0 1104 19:11:37:435
Thread exit ID: 988; Exit code: 0 988 19:11:37:435
Allocation Profiler results generated 19:11:38:014
Monitor Call Tree Details Event View Call Graph PE Reader Disassembler

AQtime integrated into Borland Developer Studio

As you can see, the Allocation profiler results are divided into two categories: **Classes** and **Objects**. These categories are displayed as subnodes of the result set node in the Explorer panel. The contents of the **Report**, **Details**, **Call Graph** and other panels depend on the currently selected category. This is described in detail below. Before proceeding, we would like you to pay attention to the four items displayed on the Profiler toolbar:

- By default, the Allocation profiler only reports about class instances (objects) that exist in the profiled application at the time of the result generation. However, if you check the **Show all loaded classes** item on the **Profiler** toolbar, AQtime will report about all classes being profiled even if no instances (objects) were created for these classes since the application started. To hide these classes from the report, uncheck the **Show all loaded classes** item.
- The profiler is able to trace memory leaks produced by your application as well as by the MFC or VCL library code, which you use to compile your application. To filter out "known" IDE memory leaks, press the **Filter standard leaks** toolbar item. AQtime will automatically detect the compiler version, which you used to create your application, and will hide the memory leaks specific to this version. Using this feature you can concentrate on the inaccuracies of your code and exclude MFC and VCL leaks (which you cannot fix) from analysis. A list of known memory leaks is available at http://www.automatedqa.com/products/aqtime/leaks/.
- The Allocation profiler traces memory allocation and de-allocation that were performed by the modules added to the Setup panel and by other modules, which these "Setup" modules use. To exclude results of non-Setup modules from the result display, enable the **View project classes only** toolbar button. To view results of all the modules, uncheck this button. Note that the button only effects the Objects category (see below) and is ignored when the Classes category is selected.

  Note that in some applications a class or memory block can be allocated and released by different modules. For instance, a string can be allocated by the main executable and released by a dynamic link library that is used by this executable. If the DLL is not included in the AQtime project, the Allocation profiler will not be able to detect the release of the string and will report a memory leak. This may cause you to think that the main executable has a memory leak, while it does not. To avoid the confusion, include the main executable and the DLLs it uses into your AQtime project.
- Besides, the Allocation profiler traces the object creation process and lists the existing objects under the **Objects** objects category. An object can be created by any executable module: no matter whether it is added to the Setup panel or not. By default all of the existing objects are displayed. However, you can press the **Filter objects by stack** button to hide those objects that were created in non-Setup modules. Once this button is pressed, AQtime scans the first lines of the creation call stack of each object and excludes it if the corresponding instruction does not belong to any of the Setup modules.

Classes Category

When this category is selected, the Report panel displays information about classes whose instances were created during the run. For memory blocks that are not object instances (for example, for memory blocks allocated with the C++ `malloc` or Delphi `GetMem` function, or the VB `ReDim` statement) the Report panel displays either *C++ native memory*, *VCL native memory* or *VB native memory* class name.

Other possible category names include --

- *Memory Overwrite Error* - This indicates that some memory-write operations violated memory block bounds. For more information on this see *Checking Bounds of Memory Blocks*
- *Committed Virtual Memory*, *Reserved Virtual Memory*, *Global heap*, *Heap*, *Heap memory*, *Local heap* - These categories contain leaks produced by calls to Windows API memory management functions. See *Tracing System Memory Management Functions*

Note that AQtime does not report the names of classes in Visual Basic 6.0 applications. These classes are traced as memory objects, and leaked classes (if any) are included in the *VB native memory* class (see *Analyzing Visual Basic 6.0 Applications*). Similary, the Allocation does not report the names of classes in Intel C++, Borland C++ and GNU CC applications. The classes in these applications are traced as memory objects and the leaks are included into the *C++ native memory class* (see *Analyzing Intel C++, Borland C++ and GNU CC Applications*).

As for classes that reside in Visual C++, Delphi and C++Builder applications, AQtime may or may not trace them as memory blocks. In order for AQtime to be able to trace a class, as a class, not as a memory block, the class must meet certain requirements. For more information, see *Analyzing Visual C++ Applications, Analyzing Delphi Applications* and *Analyzing C++Builder Applications*.

Each row in the Report panel shows profiling results for every single class: the total and current number of class instances, their size, etc. (For more detailed information, review the *Allocation Profiler - Report Panel Columns* topic). This gives you a summary view on what happened in the application during profiling (you can also view the summary results in the **Summary** panel). Note that by default the Report panel holds only some of available columns. You can add more columns to the panel or remove columns from it. For more information on this, see *Adding and Removing Columns* in AQtime help.

To determine if class instances were existing in memory at the moment of results generation, check the **Live Count** column value. If it is greater than 0, class instances existed. If you profile an unmanaged application and obtain results upon closing the application, non-zero values in the *Live Count* column help you find memory leaks. To find them quicker, you can sort or filter results on the Live Count column.

The footer of the Report panel column holds summary values for data displayed in that column. For instance, the footer of the **Live Size** column displays the summary size of all class instances that existed in memory at the moment of results generation. If you select two or more classes in the Report panel, the footer will show the summary values for the selected classes only (for more information on how to select several rows in a panel, see *Selecting Several Records in a Panel* in AQtime help).

Objects Category

When this category is selected, the Report panel displays information about class instances (objects) and memory blocks that exist in the application at the moment the results are generated. Every row in the panel holds results for a single object or a memory block. The **Object Name** column serves as the identifier of the object or memory block. For instance, the name *String.5* means the fifth `String` object created after profiling started. For memory blocks this column holds values like *C++ native memory.4* or *VCL native memory.10*. These mean the 4th memory block allocated with a C++ operator (for example, `new`) or 10th memory block allocated with a VCL memory management routine (for example, `GetMem`).

When the *Check Memory Bounds* option is enabled the panel could contain information about memory operations outside of the allocated memory block. In this case the Object Name column holds a value like *Memory Overwrite Error.15*. This means that some data was written to addresses above the upper or below the lower bound of the 15th allocated memory block.

To view all objects of a certain class, filter results on the **Class Name** column (See *Filtering Results*).

To locate the approximate line of code where the error occurred, switch to the **Creation Call Stack** page in the Details panel. For Memory Overwrite Errors this page may be empty (see *Checking Bounds of Memory Blocks*).

The **Report** panel columns are completely described in a separate topic, *Allocation Profiler - Report Panel Columns*. Here we would like you to pay attention to the **Get #** column. It displays the ordinal number of the result set within a run. For instance, if you pressed **Get Results** during the Allocation profiler run, you get two result sets: one that was generated upon pressing that button and another one that were generated upon closing the profiled application. In the first result set, in all records the Get # column will hold 1; in the second

result set this column will hold 2. You can use these values for comparison purposes. For instance, when you compare two result sets, the column will clearly tell you what objects were created or deleted between the two moments of results generation.

The Report panel is the "main" results display for objects. The **Details**, **Call Graph** and **Call Tree** panels display additional results for the object selected in the Report panel.

Note: The Allocation profiler traces and displays references to managed objects only. If you select an unmanaged object in the Report panel, the Call Graph and Call Tree panels will display only the selected object, without any links to other objects.

The **Details** panel holds three tabbed pages: **Creation Call Stack**, **References From** and **References To**. You can arrange all three pages within the Details panel as you desire. For more information on this, see the description of the Details panel.

- The contents of the **Creation Call Stack** page depends on the type of the selected row in the Report panel:
  - If the row corresponds to an object, then the page displays the stack of function calls that led to the object creation. The topmost routine in this stack is the one that created the object. Columns of the Creation Call Stack page hold information that helps you locate the routine in source code. To view the source code of a routine, simply double-click it in the call stack - AQtime will bring up the Editor panel and position the cursor on the routine's code in it. The source file of the routine must be specified in the Project Search Directories dialog. In addition, to view sources of your managed applications in the Editor, you should compile the application with debug information. See *How AQtime Profilers Use Metadata and Debug Information*.
  - If the row corresponds to the Memory Overwrite Error, then the call stack shows the sequence of function calls that led to the error detection. Errors with the memory block bounds are found only when the corresponding memory block is released or reallocated, when the application terminates or when the Get Results command is executed. The contents of the call stack depends on the situation when the error was detected. If the error was detected when a corrupted block was deleted or reallocated, the call stack for the Memory Overwrite Error will hold function calls that led to the error detection, but not to the error appearance. If the error was detected when the application terminates or when the Get Results command is executed, the call stack will be empty. See *Checking Bounds of Memory Blocks*.

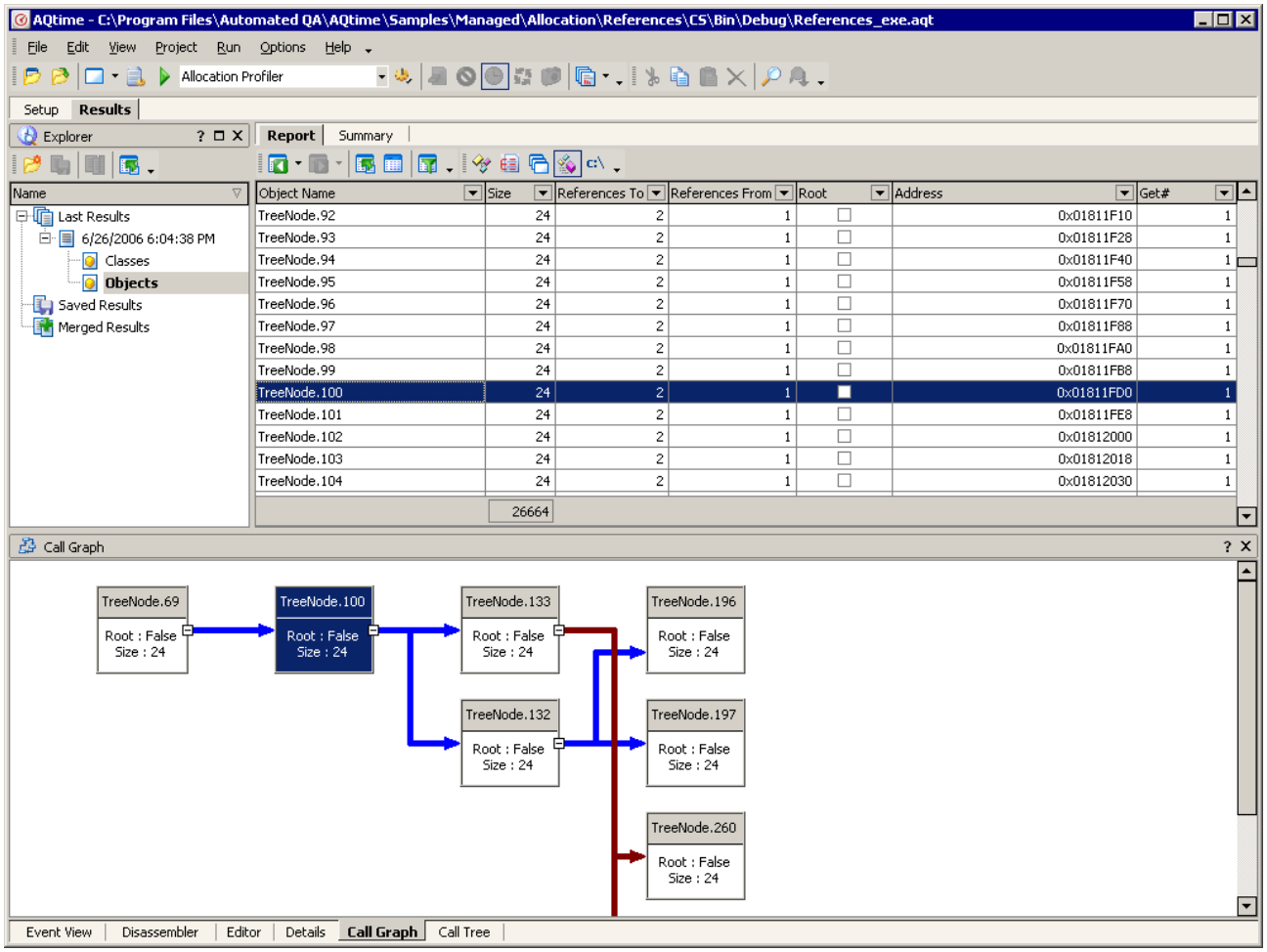
  The Creation Call Stack is available, if the profiler's *Collect stack information* option is set to *By routines* or *By lines*. To disable the call stack tracing, set this option to *None*.

  Note that sometimes the call stack may not display some information for Visual C++, Delphi or C++Builder modules. For instance, there may be no information about source files. This happens because AQtime cannot find this information in debug info. To get a more detailed call stack, you may need to recompile your application or add certain dynamic link libraries to the Setup panel. For more information on this, see *Allocation Profiler - Analyzing Visual C++ Applications*, *Allocation Profiler - Analyzing Delphi Applications* and *Allocation Profiler - Analyzing C++Builder Applications*.
- The **References To** page shows the list of objects to which the object selected in the Report panel refers. The **References From** page lists objects that refer to the object selected in the Report panel. The columns on these pages are the same as the ones in the Report panel. To view detailed information on an object, double-click it on the page. AQtime will update its panels so that they will display information relative to the new selected object.

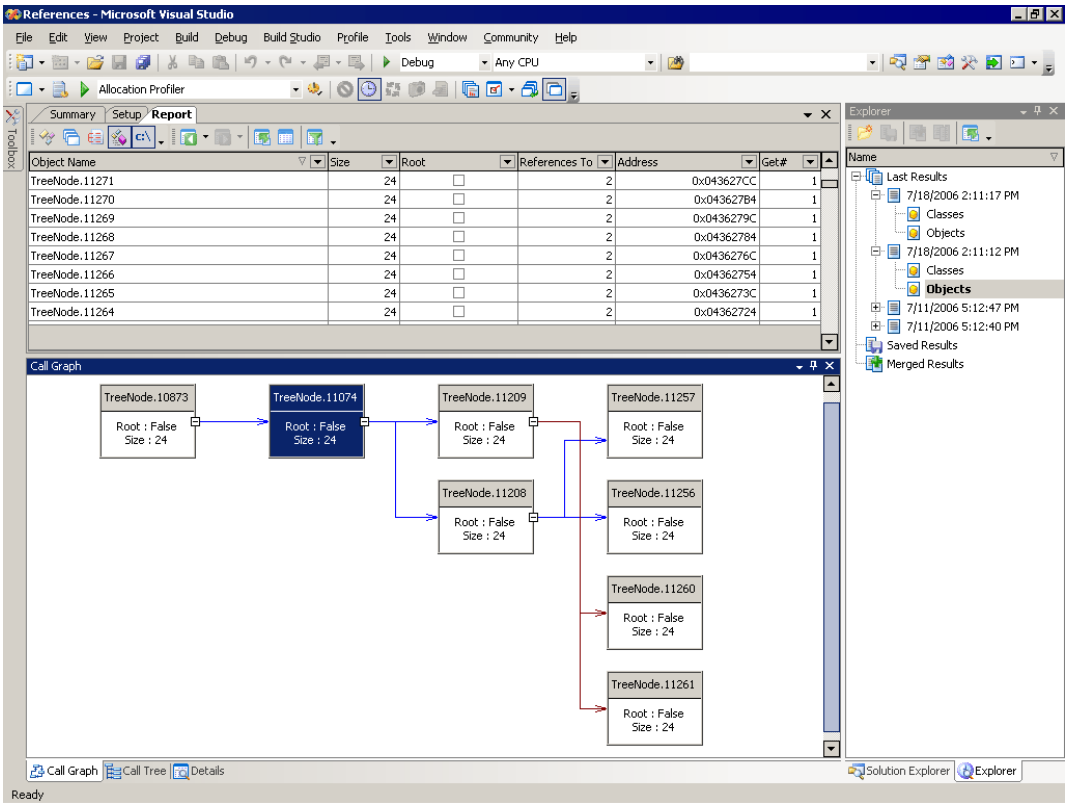
  Note: The References To and References From pages are used for analysis of managed objects only. If you selected an unmanaged object in the Report panel, these pages

will display no references.

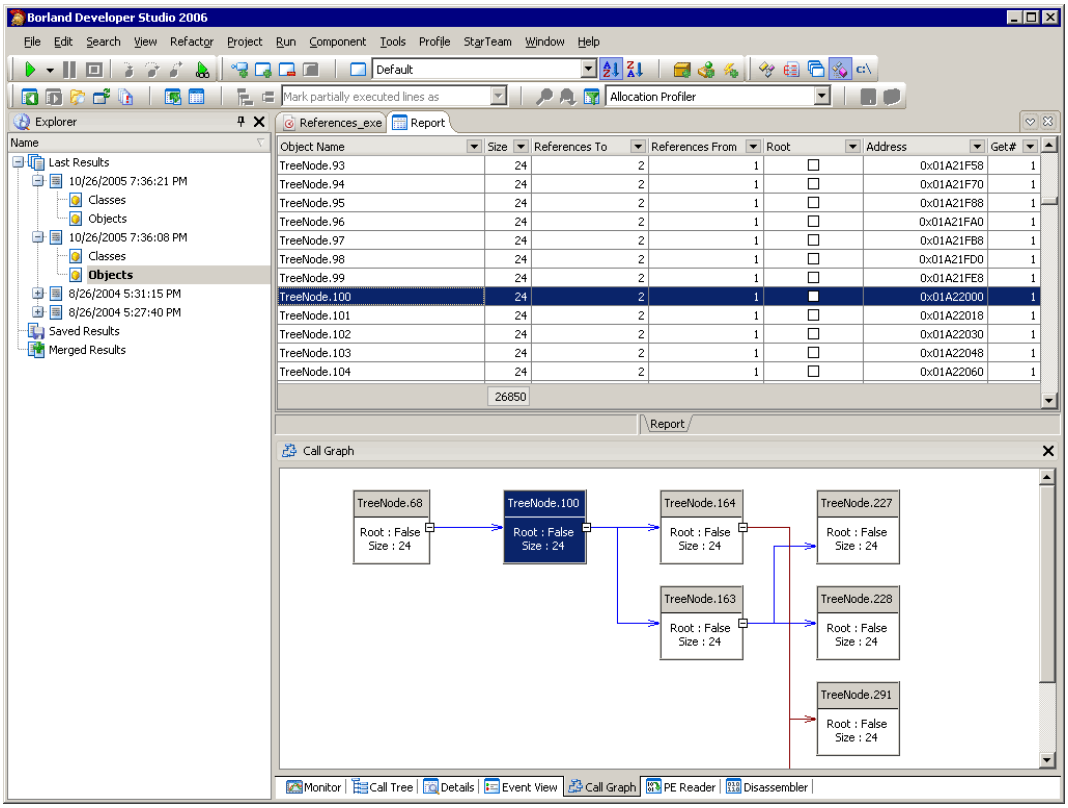
The **Call Graph** panel displays the hierarchy of object references. You can travel up and down this hierarchy by clicking the desired object.


AQtime - C:\Program Files\Automated QA\AQtime\Samples\Managed\Allocation\References\CS\Bin\Debug\References_exe.aqt
File Edit View Project Run Options Help
Allocation Profiler
Setup Results
Explorer
Report Summary
Name
Last Results
6/26/2006 6:04:38 PM
Classes
Objects
Saved Results
Merged Results
Object Name | Size | References To | References From | Root | Address | Get#
TreeNode.92 | 24 | 2 | 1 | 0x01811F10 | 1
TreeNode.93 | 24 | 2 | 1 | 0x01811F28 | 1
TreeNode.94 | 24 | 2 | 1 | 0x01811F40 | 1
TreeNode.95 | 24 | 2 | 1 | 0x01811F58 | 1
TreeNode.96 | 24 | 2 | 1 | 0x01811F70 | 1
TreeNode.97 | 24 | 2 | 1 | 0x01811F88 | 1
TreeNode.98 | 24 | 2 | 1 | 0x01811FA0 | 1
TreeNode.99 | 24 | 2 | 1 | 0x01811FB8 | 1
TreeNode.100 | 24 | 2 | 1 | 0x01811FD0 | 1
TreeNode.101 | 24 | 2 | 1 | 0x01811FE8 | 1
TreeNode.102 | 24 | 2 | 1 | 0x01812000 | 1
TreeNode.103 | 24 | 2 | 1 | 0x01812018 | 1
TreeNode.104 | 24 | 2 | 1 | 0x01812030 | 1
26664
Call Graph
TreeNode.69 Root : False Size : 24
TreeNode.100 Root : False Size : 24
TreeNode.133 Root : False Size : 24
TreeNode.196 Root : False Size : 24
TreeNode.132 Root : False Size : 24
TreeNode.197 Root : False Size : 24
TreeNode.260 Root : False Size : 24
Event View Disassembler Editor Details Call Graph Call Tree

AQtime standalone

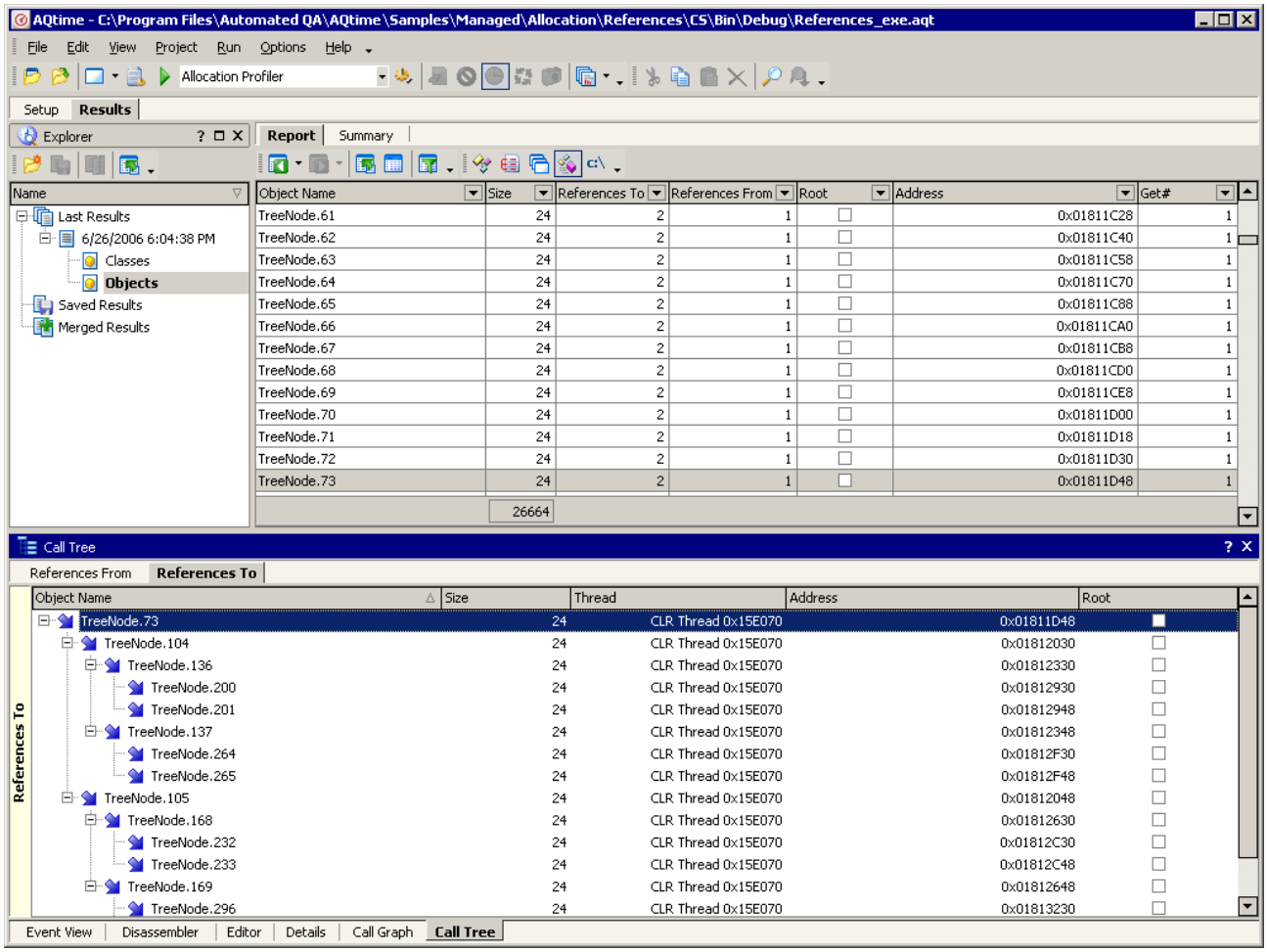


AQtime integrated into Visual Studio

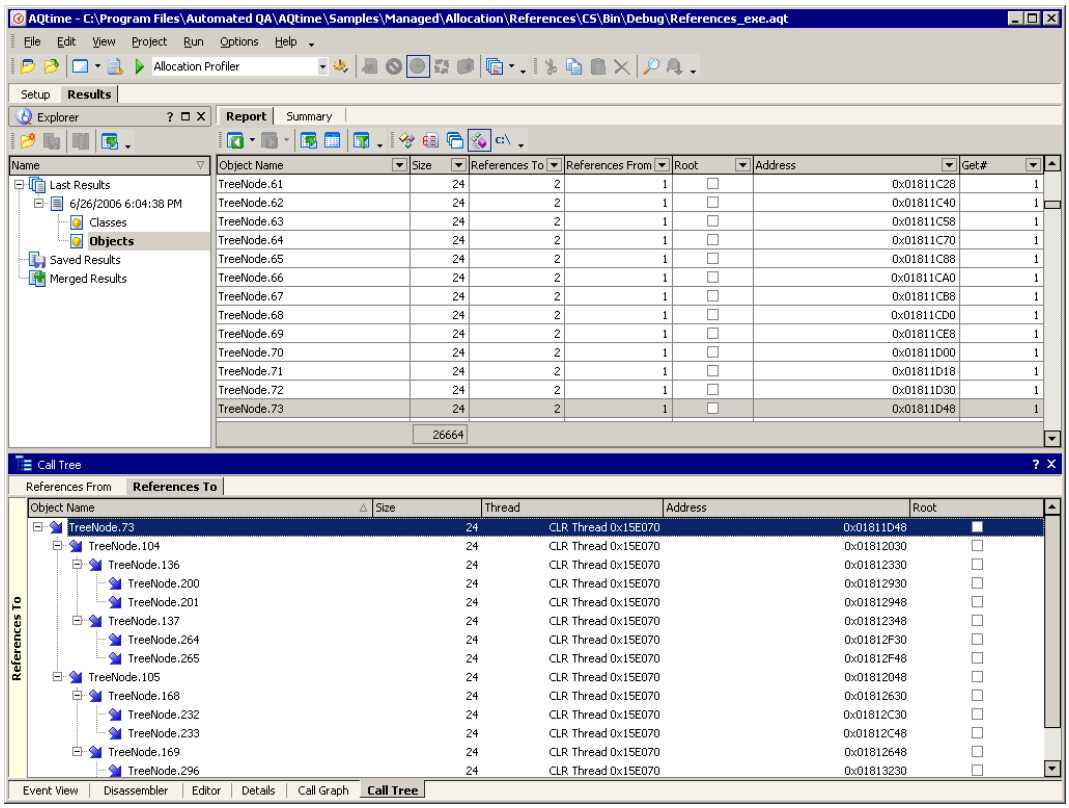


AQtime integrated into Borland Developer Studio

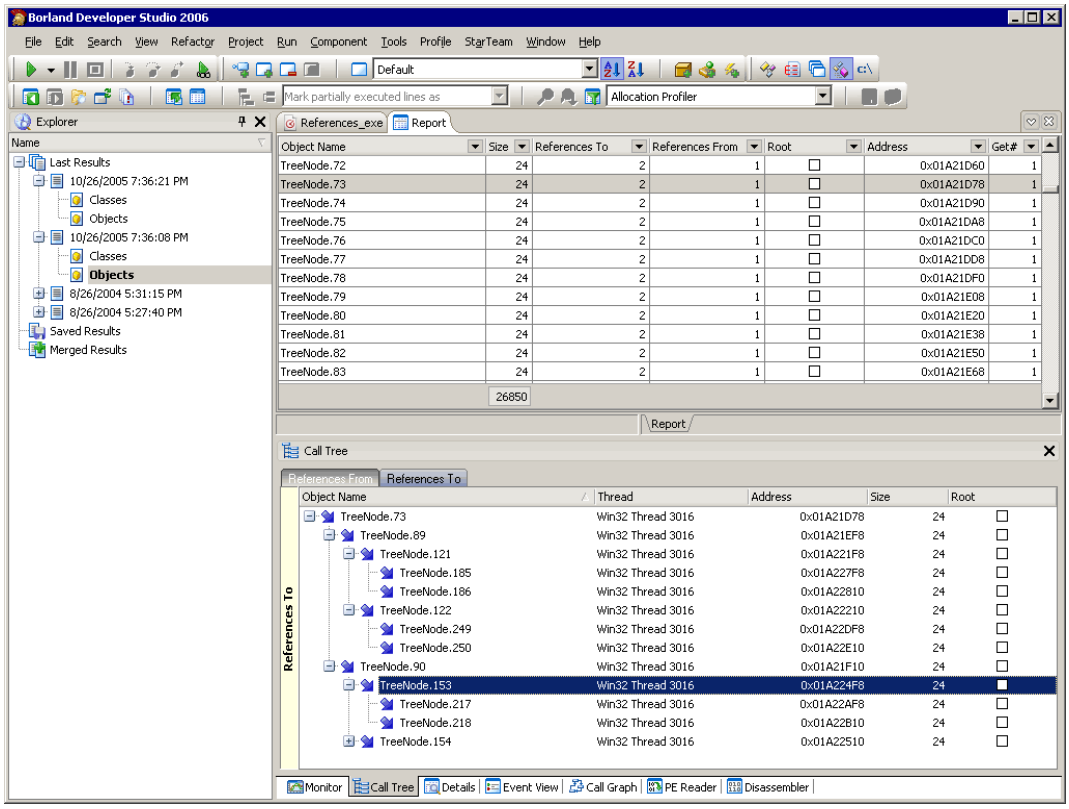
The Call Tree panel also displays the hierarchy of object references. This panel holds two pages: **References To** and **References From**. The columns in these pages are similar to the Report panel


AQtime - C:\Program Files\Automated QA\AQtime\Samples\Managed\Allocation\References\CS\Bin\Debug\References_exe.aqt
File Edit View Project Run Options Help
Allocation Profiler
Setup Results
Explorer
Report Summary
Name
Last Results
6/26/2006 6:04:38 PM
Classes
Objects
Saved Results
Merged Results
Object Name Size References To References From Root Address Get#
TreeNode.61 24 2 1 0x01811C28 1
TreeNode.62 24 2 1 0x01811C40 1
TreeNode.63 24 2 1 0x01811C58 1
TreeNode.64 24 2 1 0x01811C70 1
TreeNode.65 24 2 1 0x01811C88 1
TreeNode.66 24 2 1 0x01811CA0 1
TreeNode.67 24 2 1 0x01811CB8 1
TreeNode.68 24 2 1 0x01811CD0 1
TreeNode.69 24 2 1 0x01811CE8 1
TreeNode.70 24 2 1 0x01811D00 1
TreeNode.71 24 2 1 0x01811D18 1
TreeNode.72 24 2 1 0x01811D30 1
TreeNode.73 24 2 1 0x01811D48 1
26664
Call Tree
References From References To
References To
Object Name Size Thread Address Root
TreeNode.73 24 CLR Thread 0x15E070 0x01811D48
TreeNode.104 24 CLR Thread 0x15E070 0x01812030
TreeNode.136 24 CLR Thread 0x15E070 0x01812330
TreeNode.200 24 CLR Thread 0x15E070 0x01812930
TreeNode.201 24 CLR Thread 0x15E070 0x01812948
TreeNode.137 24 CLR Thread 0x15E070 0x01812348
TreeNode.264 24 CLR Thread 0x15E070 0x01812F30
TreeNode.265 24 CLR Thread 0x15E070 0x01812F48
TreeNode.105 24 CLR Thread 0x15E070 0x01812048
TreeNode.168 24 CLR Thread 0x15E070 0x01812630
TreeNode.232 24 CLR Thread 0x15E070 0x01812C30
TreeNode.233 24 CLR Thread 0x15E070 0x01812C48
TreeNode.169 24 CLR Thread 0x15E070 0x01812648
TreeNode.296 24 CLR Thread 0x15E070 0x01813230
Event View Disassembler Editor Details Call Graph Call Tree

AQtime standalone


AQtime - C:\Program Files\Automated QA\AQtime\Samples\Managed\Allocation\References\CS\Bin\Debug\References_exe.aqt
File Edit View Project Run Options Help
Allocation Profiler
Setup Results
Explorer
Report Summary
Name
Last Results
6/26/2006 6:04:38 PM
Classes
Objects
Saved Results
Merged Results
Object Name Size References To References From Root Address Get#
TreeNode.61 24 2 1 0x01811C28 1
TreeNode.62 24 2 1 0x01811C40 1
TreeNode.63 24 2 1 0x01811C58 1
TreeNode.64 24 2 1 0x01811C70 1
TreeNode.65 24 2 1 0x01811C88 1
TreeNode.66 24 2 1 0x01811CA0 1
TreeNode.67 24 2 1 0x01811CB8 1
TreeNode.68 24 2 1 0x01811CD0 1
TreeNode.69 24 2 1 0x01811CE8 1
TreeNode.70 24 2 1 0x01811D00 1
TreeNode.71 24 2 1 0x01811D18 1
TreeNode.72 24 2 1 0x01811D30 1
TreeNode.73 24 2 1 0x01811D48 1
26664
Call Tree
References From References To
Object Name Size Thread Address Root
TreeNode.73 24 CLR Thread 0x15E070 0x01811D48
TreeNode.104 24 CLR Thread 0x15E070 0x01812030
TreeNode.136 24 CLR Thread 0x15E070 0x01812330
TreeNode.200 24 CLR Thread 0x15E070 0x01812930
TreeNode.201 24 CLR Thread 0x15E070 0x01812948
TreeNode.137 24 CLR Thread 0x15E070 0x01812348
TreeNode.264 24 CLR Thread 0x15E070 0x01812F30
TreeNode.265 24 CLR Thread 0x15E070 0x01812F48
TreeNode.105 24 CLR Thread 0x15E070 0x01812048
TreeNode.168 24 CLR Thread 0x15E070 0x01812630
TreeNode.232 24 CLR Thread 0x15E070 0x01812C30
TreeNode.233 24 CLR Thread 0x15E070 0x01812C48
TreeNode.169 24 CLR Thread 0x15E070 0x01812648
TreeNode.296 24 CLR Thread 0x15E070 0x01813230
References To
Event View Disassembler Editor Details Call Graph Call Tree

AQtime integrated into Visual Studio


Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Default
Mark partially executed lines as
Allocation Profiler
Explorer
References_exe Report
Name
Last Results
10/26/2005 7:36:21 PM
Classes
Objects
10/26/2005 7:36:08 PM
Classes
Objects
8/26/2004 5:31:15 PM
8/26/2004 5:27:40 PM
Saved Results
Merged Results
Object Name Size References To References From Root Address Get#
TreeNode.72 24 2 1 0x01A21D60 1
TreeNode.73 24 2 1 0x01A21D78 1
TreeNode.74 24 2 1 0x01A21D90 1
TreeNode.75 24 2 1 0x01A21DA8 1
TreeNode.76 24 2 1 0x01A21DC0 1
TreeNode.77 24 2 1 0x01A21DD8 1
TreeNode.78 24 2 1 0x01A21DF0 1
TreeNode.79 24 2 1 0x01A21E08 1
TreeNode.80 24 2 1 0x01A21E20 1
TreeNode.81 24 2 1 0x01A21E38 1
TreeNode.82 24 2 1 0x01A21E50 1
TreeNode.83 24 2 1 0x01A21E68 1
26850
Report
Call Tree
References From References To
Object Name Thread Address Size Root
TreeNode.73 Win32 Thread 3016 0x01A21D78 24
TreeNode.89 Win32 Thread 3016 0x01A21EF8 24
TreeNode.121 Win32 Thread 3016 0x01A221F8 24
TreeNode.185 Win32 Thread 3016 0x01A227F8 24
TreeNode.186 Win32 Thread 3016 0x01A22810 24
TreeNode.122 Win32 Thread 3016 0x01A22210 24
TreeNode.249 Win32 Thread 3016 0x01A22DF8 24
TreeNode.250 Win32 Thread 3016 0x01A22E10 24
TreeNode.90 Win32 Thread 3016 0x01A21F10 24
TreeNode.153 Win32 Thread 3016 0x01A224F8 24
TreeNode.217 Win32 Thread 3016 0x01A22AF8 24
TreeNode.218 Win32 Thread 3016 0x01A22B10 24
TreeNode.154 Win32 Thread 3016 0x01A22510 24
References To
Monitor Call Tree Details Event View Call Graph PE Reader Disassembler

AQtime integrated into Borland Developer Studio

Allocation Profiler – Report Panel Columns

When you view results of the **Allocation** profiler, the **Report** panel contents depend on the currently selected category in the Explorer panel (see description of the Allocation profiler).

Classes Category

When this category is active, the Report panel shows information about classes, whose instances were created, and about memory blocks that were allocated after the profiling started. The panel holds the following columns:

Column	Description
Class Name	Name of the class. For memory blocks that were allocated via C++'s, Delphi's or VB's memory management routines and statements, the Class Name column holds the *C++ native memory*, *VCL native memory* or *VB native memory* value. This column may also contain the *Memory Overwrite Error* value. It indicates that the profiler detected that the application code violated bounds of memory blocks that were allocated during the application execution. See *Checking Bounds of Memory Blocks With the Allocation Profiler* for more information.
Finalizable	Specifies whether the class overrides the `Finalize` method (C# and VC++.NET use the destructor syntax for `Finalize`). This column is used only for classes defined in managed modules.
Live Count	Number of instances (objects) that currently exist in memory. For the *C++ native memory*, *VCL native memory* and *VB native memory* classes this column holds the number of allocated memory blocks that currently exist in memory. For the *Memory Overwrite Error* row, the column value specifies the number of memory block violations detected. A non-zero value in the Live Count column may indicate memory leaks. See *Searching for Memory Leaks* and *Analyzing Allocation Profiler Results.*
Live Size	The size of currently existing class instances or memory blocks (in bytes). Note that the amount of allocated memory shown in AQtime may differ from the amount of memory shown for your application in the Task Manager. See a note about this in the description of the Allocation Profiler. For the *Memory Overwrite Error* row, the column value specifies the size of all memory blocks, whose bounds were violated. A non-zero value in the Live Size column may indicate memory leaks. See *Searching for Memory Leaks* and *Analyzing Allocation Profiler Results.*
Module Name	Name of the module, where the class is defined.For the *C++ native memory*, *VCL native memory* and *VB native memory* classes, this column holds the name of the module from which the corresponding C++, VCL or VB memory management routines were called.
Namespace	Namespace that holds the class. This column is used only for classes defined in managed modules.
Peak Created	Maximum number of concurrent instances reached during the run. For the *C++ native memory*, *VCL native memory* and *VB native memory* classes this column holds the maximum number of allocated memory blocks that concurrently existed during the run.
Peak Size	Maximum size of concurrent instances reached during the run (in bytes). For the

	C++ native memory, *VCL native memory* and *VB native memory* classes this column holds the maximum size of allocated memory blocks that concurrently existed during the run.
Token	Token of the class. This column is used for classes defined in managed modules only.
Total Created	Total number of class instances (memory blocks) that were created (allocated) during the application run. For the *Memory Overwrite Error* row, the column value coincides with the Live Count column value.
Total Size	Memory needed for all the instances (memory blocks) that were created (allocated) during the run. For the Memory Overwrite Error row, the column value coincides with the Live Size column value.

Objects Category

When this category is active, the Report panel displays information about objects that were created and about memory blocks that were allocated after the profiling started. The panel holds the following columns:

Column	Description
#	The creation number of the given object or memory block.
Class Name	The name of the object's class. For memory blocks, this column contains the *C++ native memory*, *VCL native memory* or the *VB native memory* value depending on what memory management routines were used to allocate the block. This column may also contain the *Memory Overwrite Error* value. It indicates that the profiler detected that the application code violated bounds of memory blocks that were allocated during the application execution. See *Checking Bounds of Memory Blocks* for more information.
Get #	The ordinal number of the Get Results command that generated the current results set. For instance, if you pressed **Get Results** two times during the profiler run, you will get three result sets (the third will be generated after the application closes) with numbers 1, 2 and 3. The Get # value in all records of the first result set will hold 1; in the second result set this column will hold 2 and in the third result set the column will hold 3. The Get # column is used for comparison purposes. It lets you easily see which objects were created or deleted between two result generations.
Object Name	The object name. It is formed as *Class Name* + *period* + *number*. For example, *TestClass.3* means the third `TestClass` object that was created after the profiling started. Memory blocks and memory violation results are named using the same principle.
References From	The number of objects that refer to the given object. This column is used only for objects in managed applications.
References To	The number of objects to which the given object refers. This column is used only for objects in managed applications.
Root	This column is used for objects of managed applications only. If Root is checked, the object is referred to by an existing global or local variable or by a function parameter. If Root is unchecked, the object is referred to by a property of another object's field.
Size	Size of the object or memory block in bytes. For the *Memory Overwrite Error* row,

	the column displays the size of the block, whose bounds were violated.
Thread	Specifies the thread where the object's constructor was called (where the allocation routine of the memory block was called).

Allocation Profiler – Columns of the Details and Call Tree Panels

When you review the **Allocation** profiler results, the **Report** panel displays information on classes, objects and memory blocks that existed in memory at the moment of results generation (see *Allocation Profiler - Overview*). The results that are shown in the **Report**, **Details**, **Call Graph** and **Call Tree** panels depend on the category selected in the **Explorer** panel. If the *Objects* category is selected, the Report panel displays results for objects and detected memory block violations.

Note: The profiler detected violations of memory block bounds when the block is deleted or reallocated, when the application terminates or when the Get Results command is executed. If the profiler detected corrupted memory blocks when this block is deleted or reallocated, the call stack will display the sequence of function calls that led to the block deletion or reallocation. If the profiler detected the violations of memory block bounds when the application is terminated or when the Get Results command is executed, then the call stack will be empty. See *Checking Bounds of Memory Blocks*.

If you select a Report panel row that corresponds to an object, the Allocation profiler displays information about object references in the Details and Call Tree panels. Both panels contain the **References From** and **References To** panes. The References From pane lists all objects that refer to the currently selected object. The References To pane shows objects, to which the selected object refers. (The panes do not display information if you have selected a Report panel row that corresponds to a memory block violation).

The Allocation profiler traces references to managed objects only. If you select an unmanaged object in the Report panel, the Call Tree panel and the References To and References From pages of the Details panel will display only the selected object, without any links to other objects.

The information about object references is shown in grids that hold the following columns (both Call Tree and Details have the same set of columns):

Column	Description
#	The creation number of the object.
Class Name	The name of the object's class.
Count	The number of references to/from the object (for instance, if object `A` that is displayed in the Report panel holds two references to object `B`, the Count column of the References To table will hold 2 for the object `B`).
Object Name	The object name. It is formed as *Class Name* + *period* + *number*. For example, *TestClass.3* means the third `TestClass` object that was created after the profiling started.
Root	If Root is checked, the object is referred to by an existing global or local variable or by a function parameter. If Root is unchecked, the object is referred to by a property of another object's field.
Size	The object's size in bytes.
Thread	Specifies the thread where the object's constructor was called.

If the *Objects* category is selected in the Explorer panel, the Details panel also includes the **Creation Call Stack** pane. It displays the stack of function calls that led to creation of the object selected in the Report panel. The routine that created the object is at the top of the call stack. This information is shown in the grid that has the following columns:

Column	Description
Class Name	Name of the object class that holds the routine.
Module Name	Name of the module that holds the routine.
Routine Name	Name of the routine.
Source File	Name of the source file for the routine. The values for this column are read from the application's debug info.
Source Line	Source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.

Allocation Profiler Options

The **Allocation** profiler includes two groups of customizable options:

- One group contains options that affect the current result display. When you change these options, AQtime refreshes the data in its panels.
- Another group includes options that have effect on the profiler functioning. Changes in these options will only apply to future profiler runs.

To modify options that affect the result display, use items of the Profiler toolbar(by default, it is displayed at the top of the Report panel). If this toolbar is hidden, right-click somewhere in the toolbar area and select **Profiler** from the subsequent popup list. On the toolbar, the following items are available:

- **Show all loaded classes** – If this option is on, profiling results include all the classes being profiled. Otherwise, the results include only the classes whose instances had been created by the moment the results were generated. This option also affects the class results that are displayed in the **Monitor** panel for the Allocation profiler. See *Using the Monitor Panel With the Allocation Profiler*.
- **Filter standard leaks** - If your application includes code written using MFC, VCL or other libraries, there will be inevitable memory leaks due to errors in the imported library code. Using the Filter standard leaks item can hide these leaks from profiling results. If the item is pressed, AQtime excludes known memory leaks that were produced by third-party software from the profiling results. This helps you concentrate on the inaccuracies of your code and don't take in account memory problems of IDE compilers, VCL components, Microsoft Foundation Classes and others. A list of known memory leaks is available at http://www.automatedqa.com/products/aqtime/leaks/.
- **View project classes only** - If this item is pressed, AQtime only displays profiling results for those modules that are added to the Setup panel. Otherwise, it displays the results for all modules used by the profiled application.

  This filter applies both to **Classes Data** and **Objects** categories.
- **Filter objects by stack** - If this item is pressed, AQtime only anylizes the object's creation call stack and displays results for objects that were created directly by any of the modules listed in the Setup panel. Otherwise it displays results for all objects that existed when the results were generated.

This filter applies only to **Objects** category.

- **Show routines with class names** - If this item is pressed, the *Routine Name* column of the Details panel for the Allocation profiler displays the name of the given routine along with the name of the class the routine belongs to. Otherwise, this column displays only the routine name.
- **File names with path** – If this option is enabled, the **Source File** and **Module Name** columns of the **Report**, **Details**, **Call Tree** and **Monitor** panels for the Allocation profiler hold the entire path to the given file. Otherwise, these columns hold the file name only.

To modify options that have effect on the way the profiler functions, do any of the following:

- *AQtime standalone:*
  - select **Options | Options** from AQtime's main menu and then choose the **Profilers | Allocation | Allocation Profiler** group in the ensuing Options dialog;
  - press **Configure Current Profiler** on the **Standard** toolbar when the **Allocation** profiler is selected.
- *AQtime integrated into Visual Studio:*
  - select **Tools | Options** from Visual Studio's main menu and then select the **AQtime | Profilers | Allocation | Allocation Profiler** group in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - select **Profile | Options** from Borland Developer Studio's main menu and then choose the **Profilers | Allocation | Allocation Profiler** group in the ensuing Options dialog.

Options include:

- **Collect stack information** – Specifies how the profiler should collect information on call stacks when creating objects. The following values are available: *None*, *By routines* and *By lines*. Tracing the call stack can significantly slow down the profiled application. If you are only interested in objects (how many of them exist, their size, etc.), you can set this option to *None*. *By routines* means that the call stack entries will include information about routines only. If you want the call stack entries to include information on source line numbers as well, set the option to *By lines*. This will let you, for example, determine from which source line a function was called. Tracing source lines, however, requires time.
- **Check system memory allocations** - If this option is selected, AQtime traces calls to system memory management functions. Otherwise, it only traces calls to functions of the runtime memory manager. See *Tracing System Memory Management Functions*.
- **Check memory bounds** – If this option is checked, AQtime traces whether the profiled application writes to memory below or above the allocated bounds of a memory block and whether it releases the allocated memory correctly. For more information, see *Checking Bounds of Memory Blocks With the Allocation Profiler*.

  Note that if the *Check memory bounds* option is enabled then the Clear Results command is disabled. This means you cannot remove the accumulated profiling results during profiling.
- **Fill released memory blocks** - Specifies whether to overwrite the data stored in the released blocks. The actual data is replaced with the 0xDD values. This helps to reveal situations when the data is still read from the block that has already been freed. See the *Tracing Attempts to Access Released Memory* topic for details.
- **Thread model** – Specifies which thread model AQtime uses to trace the call stack for functions that allocate memory blocks. For more information on supported values, see *Profiling Multiple Threads* and *Profiling COM Logical Threads*.

BDE SQL Profiler

BDE SQL Profiler - Overview

The BDE SQL profiler lets you measure and log the execution time of SQL queries or SQL stored procedures called through the Borland Database Engine (BDE). This topic provides the BDE SQL profiler overview and describes the profiler results. The profiler description contains the following topics:

- *Overview (this topic)*
- *Description of Profiler Results (this topic)*
- *Description of the Report Panel Columns*
- *Description of the Details Panel Columns*
- *BDE SQL Profiler Options*

Overview

The execution speed of an SQL database application depends directly on the speed of SQL queries. The BDE SQL profiler times the execution of SQL queries and SQL stored procedures when commanded through the Borland Database Engine.

The BDE SQL profiler works with applications compiled with Borland Delphi v. 3 - 7, 2005 - 2007 and C++Builder v. 3 - 6, 2006. It tracks calls to the `CreateCursor` and `Prepare` methods of the `TQuery` object and calls to the `ExecProc` and `Prepare` methods of the `TStoredProc` object (`CreateCursor` is called from the `Open` and `ExecSQL` methods).

The profiler does not support BDE.NET components. It ignores queries that are performed through BDE.NET and does not display information about them in the profiling results.

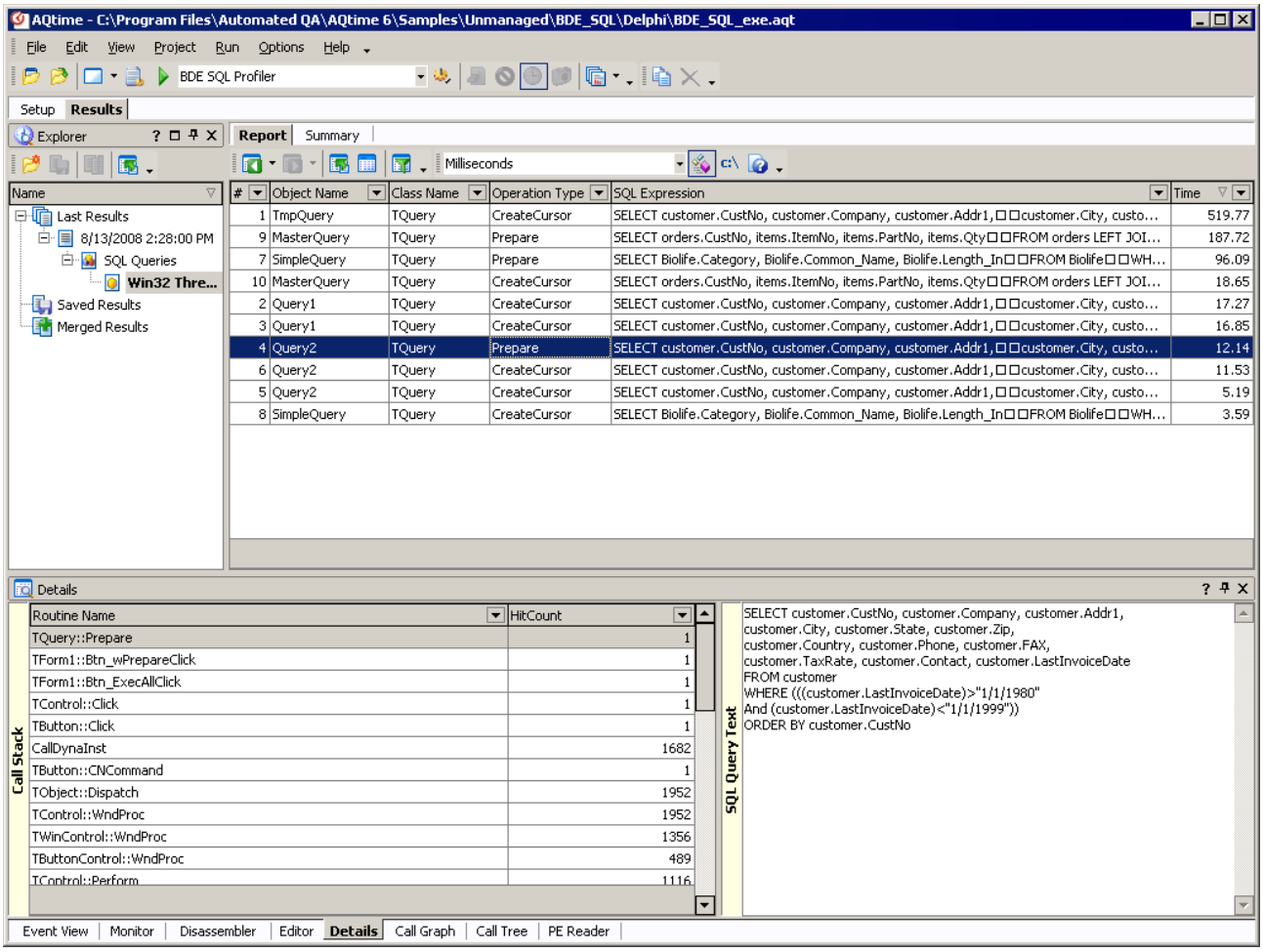
Note: If you use a computer that has several processors or a multiple-core processor (for example, dual-core CPU) and has Windows XP Service Pack 2, then you must install the Windows update #896256 in order for the profiler to be able to time your application correctly. The update is available on Microsoft's web site:

- http://support.microsoft.com/kb/896256

BDE SQL Profiler - Description of Profiler Results

Brief results of the BDE SQL profiler are displayed in the Summary panel. It shows the total number of executed SQL queries and the worst performing queries. Information on individual SQL queries is displayed in the Report panel.

Here is an example of the BDE SQL profiler output:


AQtime - C:\Program Files\Automated QA\AQtime 6\Samples\Unmanaged\BDE_SQL\Delphi\BDE_SQL_exe.aqt
File Edit View Project Run Options Help
BDE SQL Profiler
Setup Results
Explorer
Report Summary
Milliseconds
Name
Last Results
8/13/2008 2:28:00 PM
SQL Queries
Win32 Thre...
Saved Results
Merged Results
Object Name Class Name Operation Type SQL Expression Time
1 TmpQuery TQuery CreateCursor SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 519.77
9 MasterQuery TQuery Prepare SELECT orders.CustNo, items.ItemNo, items.PartNo, items.Qty□□FROM orders LEFT JOI... 187.72
7 SimpleQuery TQuery Prepare SELECT Biolife.Category, Biolife.Common_Name, Biolife.Length_In□□FROM Biolife□□WH... 96.09
10 MasterQuery TQuery CreateCursor SELECT orders.CustNo, items.ItemNo, items.PartNo, items.Qty□□FROM orders LEFT JOI... 18.65
2 Query1 TQuery CreateCursor SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 17.27
3 Query1 TQuery CreateCursor SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 16.85
4 Query2 TQuery Prepare SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 12.14
6 Query2 TQuery CreateCursor SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 11.53
5 Query2 TQuery CreateCursor SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 5.19
8 SimpleQuery TQuery CreateCursor SELECT Biolife.Category, Biolife.Common_Name, Biolife.Length_In□□FROM Biolife□□WH... 3.59
Details
Call Stack
Routine Name HitCount
TQuery::Prepare 1
TForm1::Btn_wPrepareClick 1
TForm1::Btn_ExecAllClick 1
TControl::Click 1
TButton::Click 1
CallDynaInst 1682
TButton::CNCommand 1
TObject::Dispatch 1952
TControl::WndProc 1952
TWinControl::WndProc 1356
TButtonControl::WndProc 489
SQL Query Text
SELECT customer.CustNo, customer.Company, customer.Addr1,
customer.City, customer.State, customer.Zip,
customer.Country, customer.Phone, customer.FAX,
customer.TaxRate, customer.Contact, customer.LastInvoiceDate
FROM customer
WHERE (((customer.LastInvoiceDate)>"1/1/1980"
And (customer.LastInvoiceDate)<"1/1/1999"))
ORDER BY customer.CustNo
Event View Monitor Disassembler Editor Details Call Graph Call Tree PE Reader

As you can see, each row of the Report panel contains the results of the `ExecSQL`, `ExecProc` or `Prepare` method execution. The panel columns indicate the class name and the name of the query or stored procedure, a SQL expression and other information. To obtain the execution time of a query or stored procedure, view the **Time** column. For complete information on the Report panel columns, see *BDE SQL Profiler - Report Panel Columns*.

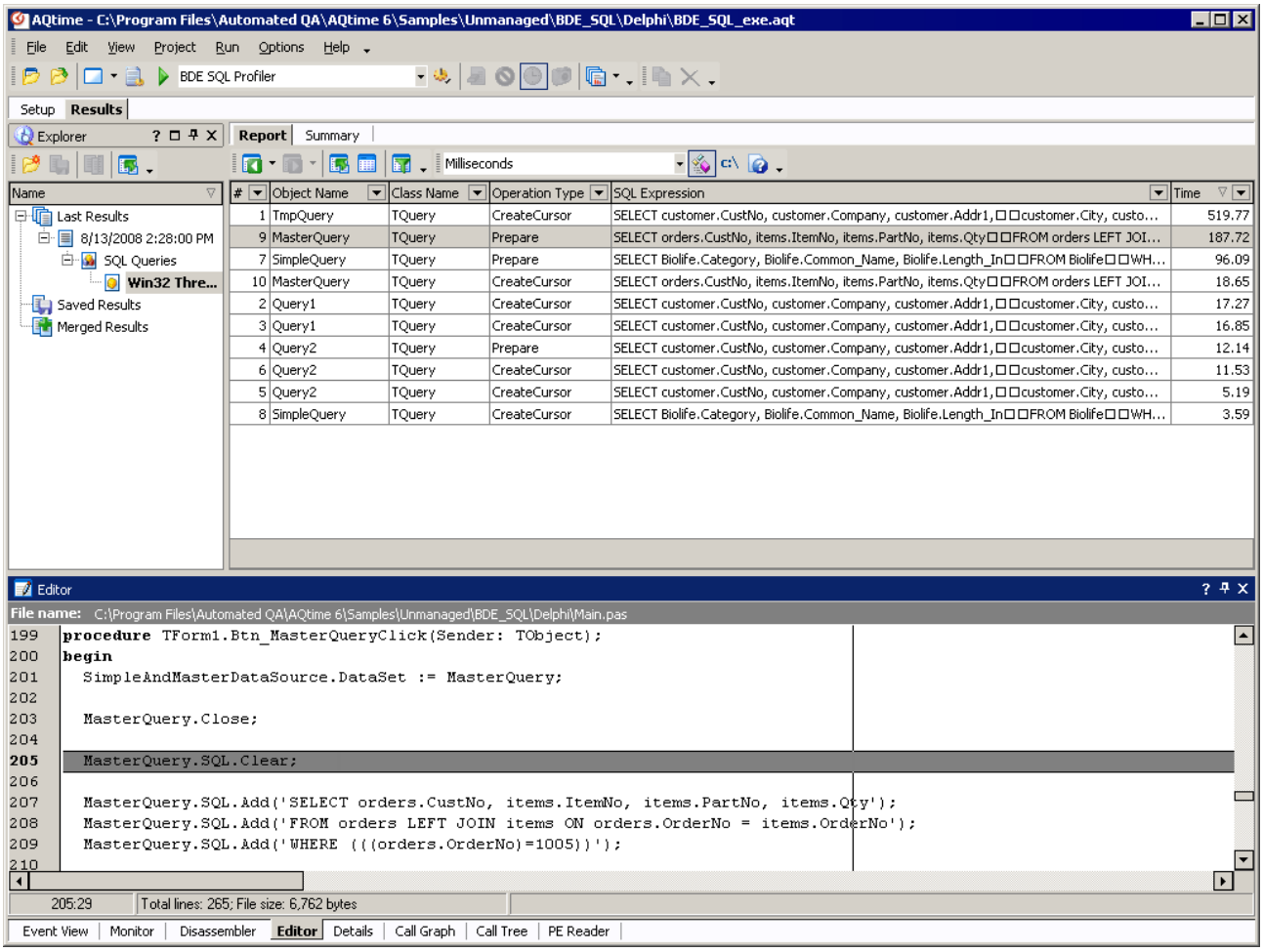
Note that by default, the BDE SQL profiler shows all available columns of the Report panel. You can remove, add and arrange the columns in the panel. For more information, see the *Adding and Removing Columns* and *Arranging Columns, Lines and Panels* topics.

Each row in the Report panel corresponds to a single SQL query in your application. Clicking on a query in the Report panel will update the contents of the Details panel, so it will display information concerning that query.

The Detail panel contains the **Call Stack** pane that displays the sequence of functions that call the BDE operation currently selected in the Report panel.

The complete code of the selected query (in addition to the SQL Expression column of the Report panel) is shown in the **SQL Query Text** pane on the right.

Double-clicking a row in the Details table will move the cursor in the Editor panel to the source code line for the compliance routine (the path to the source files must be specified in the Project Search Directories and Search Directories dialogs).


AQtime - C:\Program Files\Automated QA\AQtime 6\Samples\Unmanaged\BDE_SQL\Delphi\BDE_SQL_exe.aqt
File Edit View Project Run Options Help
BDE SQL Profiler
Setup Results
Explorer
Report Summary
Milliseconds
Name
Last Results
8/13/2008 2:28:00 PM
SQL Queries
Win32 Thre...
Saved Results
Merged Results
Object Name Class Name Operation Type SQL Expression Time
1 TmpQuery TQuery CreateCursor SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 519.77
9 MasterQuery TQuery Prepare SELECT orders.CustNo, items.ItemNo, items.PartNo, items.Qty□□FROM orders LEFT JOI... 187.72
7 SimpleQuery TQuery Prepare SELECT Biolife.Category, Biolife.Common_Name, Biolife.Length_In□□FROM Biolife□□WH... 96.09
10 MasterQuery TQuery CreateCursor SELECT orders.CustNo, items.ItemNo, items.PartNo, items.Qty□□FROM orders LEFT JOI... 18.65
2 Query1 TQuery CreateCursor SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 17.27
3 Query1 TQuery CreateCursor SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 16.85
4 Query2 TQuery Prepare SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 12.14
6 Query2 TQuery CreateCursor SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 11.53
5 Query2 TQuery CreateCursor SELECT customer.CustNo, customer.Company, customer.Addr1,□□customer.City, custo... 5.19
8 SimpleQuery TQuery CreateCursor SELECT Biolife.Category, Biolife.Common_Name, Biolife.Length_In□□FROM Biolife□□WH... 3.59
Editor
File name: C:\Program Files\Automated QA\AQtime 6\Samples\Unmanaged\BDE_SQL\Delphi\Main.pas
199 procedure TForm1.Btn_MasterQueryClick(Sender: TObject);
200 begin
201 SimpleAndMasterDataSource.DataSet := MasterQuery;
202
203 MasterQuery.Close;
204
205 MasterQuery.SQL.Clear;
206
207 MasterQuery.SQL.Add('SELECT orders.CustNo, items.ItemNo, items.PartNo, items.Qty');
208 MasterQuery.SQL.Add('FROM orders LEFT JOIN items ON orders.OrderNo = items.OrderNo');
209 MasterQuery.SQL.Add('WHERE (((orders.OrderNo)=1005))');
210
205:29
Total lines: 265; File size: 6,762 bytes
Event View Monitor Disassembler Editor Details Call Graph Call Tree PE Reader

BDE SQL Profiler - Report Panel Columns

When you review the BDE SQL profiler results, the Report panel contains the following columns:

Columns (in alphabetical order)	**Description**
#	The ordinal number of the query or stored procedure.
Class Name	The class name of the query or stored procedure. Normally, this is TQuery or TStoredProc.
Object Name	The name of the object that represents the query or stored procedure.
Operation Type	The type of database operation. This can be one of the following: • TQuery.CreateCursor (this operation is performed within the TQuery Open or ExecSQL methods) • TQuery.Prepare • TStoredProc.ExecProc • TStoredProc.Prepare
SQL Expression	The code of the executed SQL query. This field is empty for stored procedures.

Time	The execution time of the query or stored procedure. This does not include the data fetch time.

By default, the BDE SQL profiler shows all of these columns in the Report panel. You can remove columns from or add them to the Report panel. For more information, see the *Adding and Removing Columns* topic.

BDE SQL Profiler - Details Panel Columns

When the BDE SQL profiler displays profiling results, each row of the Report panel contains the results of a database operation execution. The BDE SQL profiler uses the Details panel to display information on calls to BDE operations. The Detail panel contains the Call Stack pane that displays the sequence of functions that call the BDE operation currently selected in the Report panel. Each line below shows the caller of the line above it. The Call Stack table contains the following columns:

Columns (in alphabetical order)	**Description**
Class Name	The name of the class to which the routine belongs.
Hit Count	The number of times the routine was called during the profiler run.
Module Name	The name of the executable module that contains the routine.
Routine Name	The name of the routine that lead to the selected BDE operation.
Source File	The name of the routine's source file. The values for this column are read from the application's debug info.
Source Line	The number of the source file's line where the routine's implementation begins.

On the right of the Details panel there is an **SQL Query Text** pane that displays the complete code of the query, selected in the Report panel. It can be useful because SQL queries can sometimes have long code that can not be fully displayed in the SQL Expression column of the Report panel.

BDE SQL Profiler Options

The BDE SQL profiler includes two groups of customizable options:

- One group holds options that affect the currently displayed results. If you change any of these options, the data contained in AQtime's panels will be refreshed.
- Another group contains options that affect how the the profiler works. If you change these options, the changes will affect future profiler runs.

Options that modify the displayed results correspond to the items of the **Profiler** toolbar. You can display this toolbar by right-clicking somewhere within the toolbar and selecting **Profiler** from the popup list. The following items are located on the toolbar:

- **Counter unit** - The *Counter unit* item lets you specify the unit of measurement for the time columns in the AQtime panel. *Seconds*, *Milliseconds*, *Microseconds* and *Machine Cycles* are available values that you can set.
- **Show routines with class names** - If this option is enabled, the **Routine Name** column of the BDE SQL profiler's Details panel holds both the class name and the routine name. Otherwise, this column only holds the routine name.

© 2010 AutomatedQA Corp.

- **File names with path** - If this option is enabled, the **Module Name** column of the BDE SQL profiler's Details panel holds the entire path to the given file. Otherwise, this column only holds the file name.

To modify options that have effect on the way the profiler functions, do any of the following:

- *AQtime standalone:*
  - select **Options | Options** from the main menu and then choose **Profilers | Performance | BDE SQL Profiler** on the left of the ensuing Options dialog;
  - press **Configure Current Profiler** on the **Standard** toolbar when the **BDE SQL** profiler is selected.
- *AQtime integrated into Visual Studio:*
  - select **Tools | Options** from Visual Studio's main menu and then choose **AQtime | Profilers | Performance | BDE SQL Profiler** from the tree view on the left of the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - select **Profile | Options** from Borland Developer Studio's main menu and then choose **Profilers | Performance | BDE SQLProfiler** on the left of the ensuing Options dialog.

There is only one option in this group:

- **Thread model** - Specifies how the BDE SQL profiler gathers data for threads in the profiled application. For more information on the available values for this option, see *Profiling Multiple Threads*.

Coverage Profiler

Coverage Profiler - Overview

The **Coverage** profiler lets you determine whether a routine or a line was executed during the profiler run and how many times it was executed. This topic provides the Coverage profiler overview and describes the profiling results. The complete profiler description includes the following sections:

- *Overview (this topic)*
- *Description of Profiler Results (this topic)*
- *Description of the Report Panel Columns*
- *Description of the Details Panel Columns*
- *Coverage Profiler Options*

Overview

The **Coverage** profiler tracks one thing: whether a routine or a line was called during the run. This lets you keep track of untested code as testing progresses over time. It may also let you find unnecessary code that you can remove, once you see that the method or line remains unexecuted under all possible conditions.

The Coverage profiler analyzes the application code (32-bit and 64-bit) at two levels of detail: **routine** and **line**. To profile the lines of a routine, you should simply add this routine to a line-level area (see *Profiling Levels*). Note that to profile managed routines at line level, you have to compile the application with debug information. See *How AQtime Profilers Use Metadata and Debug Information*. If you need to track all lines covered or not covered, begin by using the Coverage profiler on the *Full Check* area. This will let you focus on

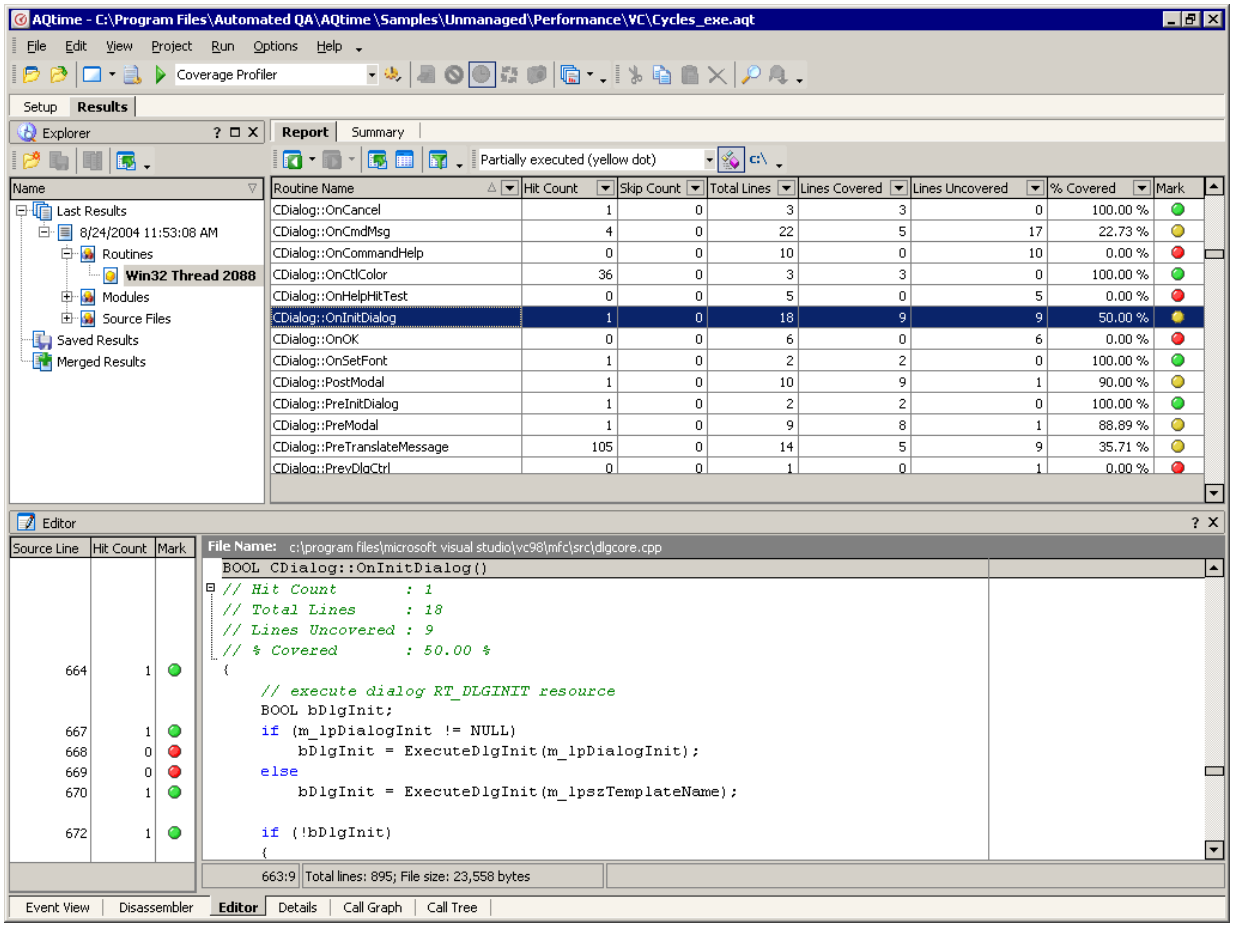
the problem files first, and then you can narrow the analysis to these files and use the Coverage profiler with them to drill down further.

The Coverage profiler also supports triggers and actions. They allow you to turn the profiling on or off exactly when it is needed during the application run. For more information, see *Using Triggers* and *Using Actions*.

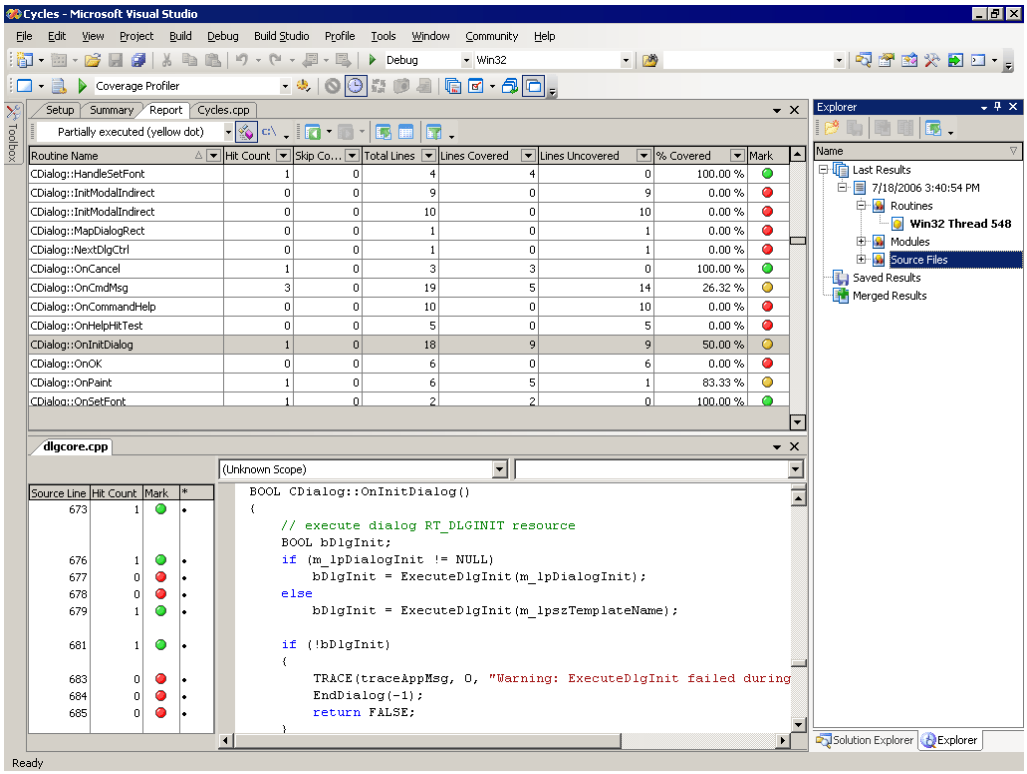
After you have run Coverage several times, you can merge profiling results to get mass statistics. Merging can be executed directly from the context menu of the **Explorer** panel (using the **Merge** item) or done automatically in the background after each profiling run (using the *Auto-merge* option of the Explorer panel). You can also compare results of several Coverage runs in order to see the changes. For more information on comparison and merge of results, see *Comparing and Merging Results*.

Coverage Profiler Results - Description of Profiler Results

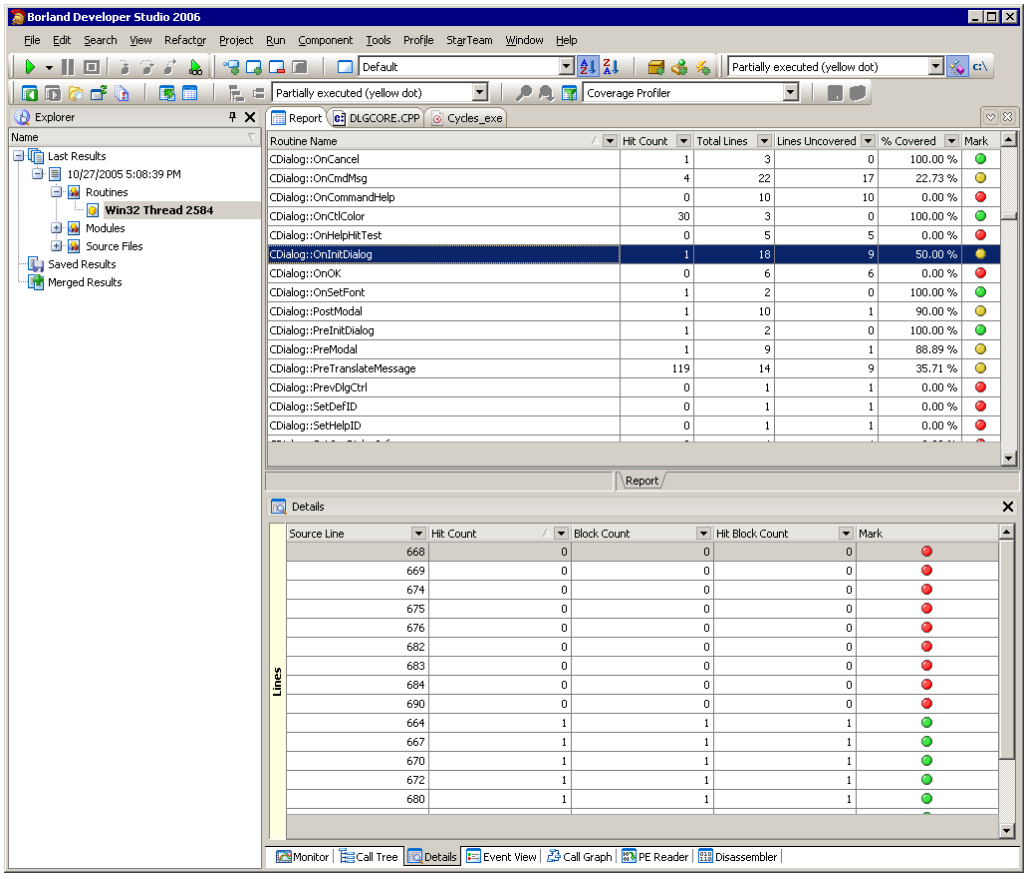
Brief results of the Coverage profiler are displayed in the **Summary** panel. It shows ten routines that were called most often than other routines and ten routines that were covered less than other application routines. Information for individual application routines, source files and modules are displayed in the **Report** panel. Here is an example of a Coverage profiler output:


AQtime - C:\Program Files\Automated QA\AQtime\Samples\Unmanaged\Performance\VC\Cycles_exe.aqt
File Edit View Project Run Options Help
Coverage Profiler
Setup Results
Explorer
Report Summary
Partially executed (yellow dot)
Name
Last Results
8/24/2004 11:53:08 AM
Routines
Win32 Thread 2088
Modules
Source Files
Saved Results
Merged Results
Routine Name Hit Count Skip Count Total Lines Lines Covered Lines Uncovered % Covered Mark
CDialog::OnCancel 1 0 3 3 0 100.00 %
CDialog::OnCmdMsg 4 0 22 5 17 22.73 %
CDialog::OnCommandHelp 0 0 10 0 10 0.00 %
CDialog::OnCtlColor 36 0 3 3 0 100.00 %
CDialog::OnHelpHitTest 0 0 5 0 5 0.00 %
CDialog::OnInitDialog 1 0 18 9 9 50.00 %
CDialog::OnOK 0 0 6 0 6 0.00 %
CDialog::OnSetFont 1 0 2 2 0 100.00 %
CDialog::PostModal 1 0 10 9 1 90.00 %
CDialog::PreInitDialog 1 0 2 2 0 100.00 %
CDialog::PreModal 1 0 9 8 1 88.89 %
CDialog::PreTranslateMessage 105 0 14 5 9 35.71 %
CDialog::PrevDlgCtrl 0 0 1 0 1 0.00 %
Editor
Source Line Hit Count Mark
File Name: c:\program files\microsoft visual studio\vc98\mfc\src\dlgcore.cpp
BOOL CDialog::OnInitDialog()
// Hit Count : 1
// Total Lines : 18
// Lines Uncovered : 9
// % Covered : 50.00 %
{
// execute dialog RT_DLGINIT resource
BOOL bDlgInit;
if (m_lpDialogInit != NULL)
bDlgInit = ExecuteDlgInit(m_lpDialogInit);
else
bDlgInit = ExecuteDlgInit(m_lpszTemplateName);
if (!bDlgInit)
663:9 Total lines: 895; File size: 23,558 bytes
Event View Disassembler Editor Details Call Graph Call Tree

AQtime standalone


Cycles - Microsoft Visual Studio
File Edit View Project Build Debug Build Studio Profile Tools Window Community Help
Debug
Win32
Coverage Profiler
Setup Summary Report Cycles.cpp
Partially executed (yellow dot)
Routine Name Hit Count Skip Co... Total Lines Lines Covered Lines Uncovered % Covered Mark
CDialog::HandleSetFont 1 0 4 4 0 100.00 %
CDialog::InitModalIndirect 0 0 9 0 9 0.00 %
CDialog::InitModalIndirect 0 0 10 0 10 0.00 %
CDialog::MapDialogRect 0 0 1 0 1 0.00 %
CDialog::NextDlgCtrl 0 0 1 0 1 0.00 %
CDialog::OnCancel 1 0 3 3 0 100.00 %
CDialog::OnCmdMsg 3 0 19 5 14 26.32 %
CDialog::OnCommandHelp 0 0 10 0 10 0.00 %
CDialog::OnHelpHitTest 0 0 5 0 5 0.00 %
CDialog::OnInitDialog 1 0 18 9 9 50.00 %
CDialog::OnOK 0 0 6 0 6 0.00 %
CDialog::OnPaint 1 0 6 5 1 83.33 %
CDialog::OnSetFont 1 0 2 2 0 100.00 %
dlgcore.cpp
(Unknown Scope)
Source Line Hit Count Mark
BOOL CDialog::OnInitDialog()
{
// execute dialog RT_DLGINIT resource
BOOL bDlgInit;
if (m_lpDialogInit != NULL)
bDlgInit = ExecuteDlgInit(m_lpDialogInit);
else
bDlgInit = ExecuteDlgInit(m_lpszTemplateName);
if (!bDlgInit)
{
TRACE(traceAppMsg, 0, "Warning: ExecuteDlgInit failed during
EndDialog(-1);
return FALSE;
Explorer
Name
Last Results
7/18/2006 3:40:54 PM
Routines
Win32 Thread 548
Modules
Source Files
Saved Results
Merged Results
Solution Explorer Explorer
Ready

AQtime integrated into Visual Studio

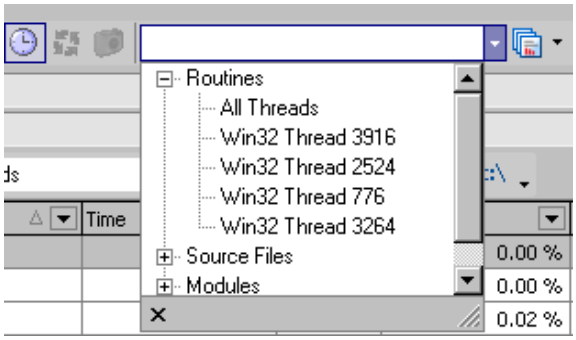

Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Default
Partially executed (yellow dot)
Coverage Profiler
Explorer
Name
Last Results
10/27/2005 5:08:39 PM
Routines
Win32 Thread 2584
Modules
Source Files
Saved Results
Merged Results
Report DLGCORE.CPP Cycles_exe
Routine Name Hit Count Total Lines Lines Uncovered % Covered Mark
CDialog::OnCancel 1 3 0 100.00 %
CDialog::OnCmdMsg 4 22 17 22.73 %
CDialog::OnCommandHelp 0 10 10 0.00 %
CDialog::OnCtlColor 30 3 0 100.00 %
CDialog::OnHelpHitTest 0 5 5 0.00 %
CDialog::OnInitDialog 1 18 9 50.00 %
CDialog::OnOK 0 6 6 0.00 %
CDialog::OnSetFont 1 2 0 100.00 %
CDialog::PostModal 1 10 1 90.00 %
CDialog::PreInitDialog 1 2 0 100.00 %
CDialog::PreModal 1 9 1 88.89 %
CDialog::PreTranslateMessage 119 14 9 35.71 %
CDialog::PrevDlgCtrl 0 1 1 0.00 %
CDialog::SetDefID 0 1 1 0.00 %
CDialog::SetHelpID 0 1 1 0.00 %
Report
Details
Source Line Hit Count Block Count Hit Block Count Mark
Lines
Monitor Call Tree Details Event View Call Graph PE Reader Disassembler

AQtime integrated into Borland Developer Studio

As you can see, the results are organized into three categories: **Routines**, **Source Files** and **Modules**. *Source Files* and *Modules* let you view summary profiling results for each source file and module in your application. The *Routines* category contains results for each single routine that was included in profiling tasks.

Within the categories the results are grouped by thread. There is also the *All threads* group that show profiling results for all threads. Note that the Coverage profiler includes the *Thread model* option that specifies how the profiler gathers statistics for threads in the profiled application. For more information on available values for this option, see *Profiling Multiple Threads*. You can change this option either using **Coverage Profiler Options** dialog, or using the **Run Settings** dialog that is shown upon starting the profiler run.

The categories are displayed as subnodes of the result set node in the **Explorer** panel. You can also select the desired category and thread from the Result Items box on the Standard toolbar:



The contents of the other panels (**Report**, **Details**, etc.) depend on the currently selected category.

- If you select the *Routines* category, the Report panel will display profiling results one routine per line. Line coverage results will be shown in the **Lines** page of the Details panel and in the Editor's grid.
- If you select the *Modules* or *Source Files* category, each row in the Report panel will display profiling results for one module (or source file). The Details and Editor panels will not be used.

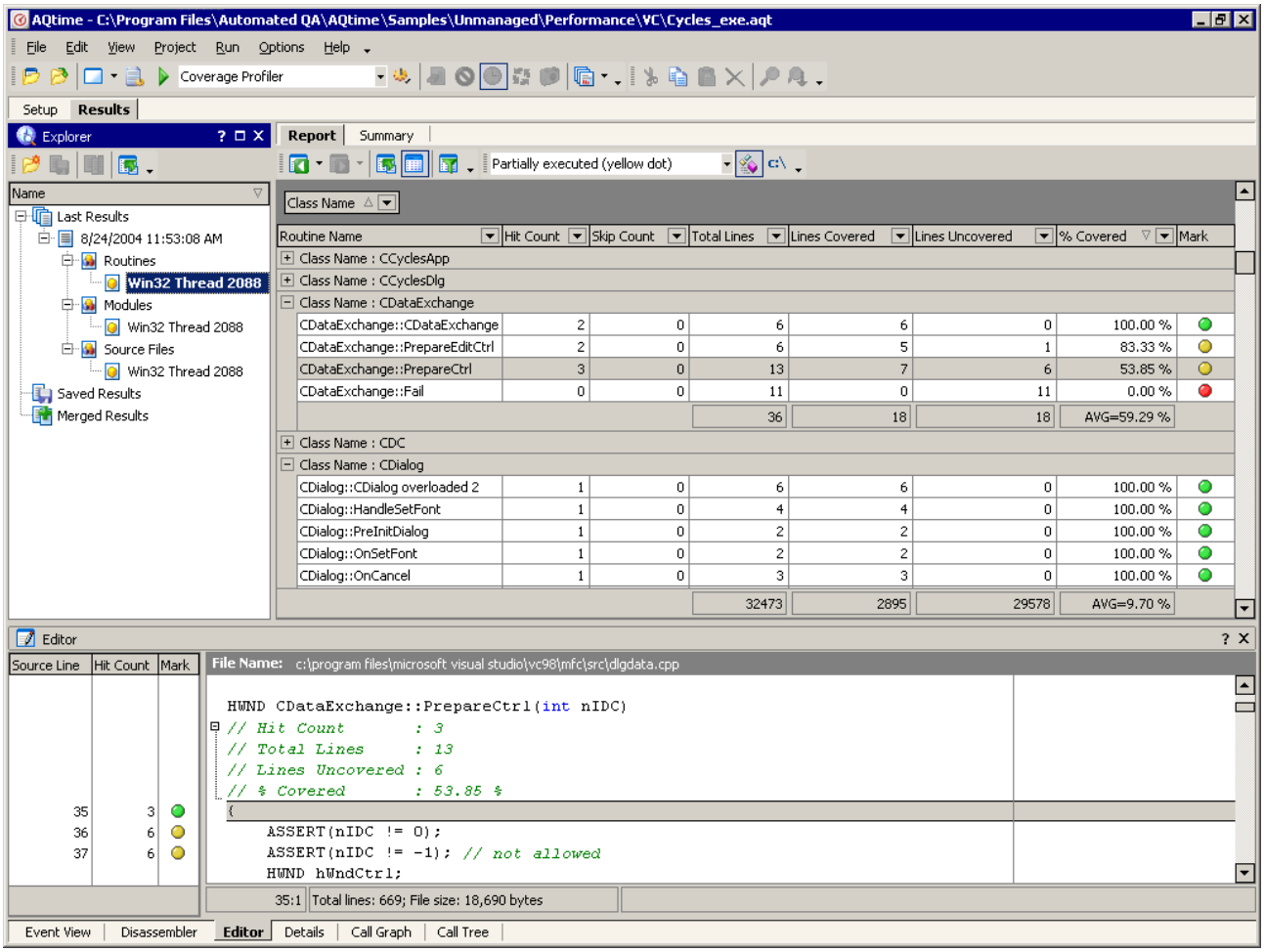
Coverage Profiler Results - Report Panel

The Report panel displays profiling results according to the category and thread selected in the Explorer panel or in the **Result Items** box on the Standard toolbar. The **Hit Count** column helps you quickly see which routines were executed. If a routine was not called during the profiler run, the Hit Count column holds 0. Otherwise, it shows how many times the routine was called. You can also use the **Lines Covered**, **Lines Uncovered**, **Total Lines** and **% Covered** columns to identify untested code. For instance, if the function includes a large number of lines, of which only a small percentage were executed during a seemingly "complete" test for the function, you might want to examine the function's algorithm. Of course, AQtime gathers line coverage statistics for those routines that were added to line-level profiling areas. If a routine was profiled at routine level, *% Covered* is either 100%, or 0%.

You can quickly filter out the routines whose source code lines were covered partially (less than a particular percentage). To do this, use the predefined result views *Routines covered less than %nn*. On the other hand, using the *Unexecuted routines only* predefined view, you can display the routines that were not executed at all. You can select any of these views from the **Result Views** dropdown list on the Standard toolbar, from the **View | Result View** menu or from the **Result Views** dialog. To display it, choose **Profile | Result Views** from Visual Studio's menu or click **Results Views** on Visual Studio's **View.AQtime** toolbar. See *Result Views*.

The **Mark** column graphically represents the Coverage result. It holds green dots for those routines that were executed during the profiler run and red dots for those routines that were not executed. If a routine was partially executed, the Mark column shows a yellow dot. For complete information on columns, see *Coverage Profiler - Report Panel Columns*.

The summary values of the **Lines Covered** and **% Covered** columns display the total number of covered lines and coverage percentage. To find the percent of covered lines in a class, unit or source file, simply group results in the Report panel by the **Class Name**, **Unit Name** or **Source File** columns. The group summary will display the coverage results for each class, unit or source file:


AQtime - C:\Program Files\Automated QA\AQtime\Samples\Unmanaged\Performance\VC\Cycles_exe.aqt
File Edit View Project Run Options Help
Coverage Profiler
Setup Results
Explorer
Report Summary
Partially executed (yellow dot)
Name
Last Results
8/24/2004 11:53:08 AM
Routines
Win32 Thread 2088
Modules
Win32 Thread 2088
Source Files
Win32 Thread 2088
Saved Results
Merged Results
Class Name
Routine Name | Hit Count | Skip Count | Total Lines | Lines Covered | Lines Uncovered | % Covered | Mark
Class Name : CCyclesApp
Class Name : CCyclesDlg
Class Name : CDataExchange
CDataExchange::CDataExchange 2 0 6 6 0 100.00 %
CDataExchange::PrepareEditCtrl 2 0 6 5 1 83.33 %
CDataExchange::PrepareCtrl 3 0 13 7 6 53.85 %
CDataExchange::Fail 0 0 11 0 11 0.00 %
36 18 18 AVG=59.29 %
Class Name : CDC
Class Name : CDialog
CDialog::CDialog overloaded 2 1 0 6 6 0 100.00 %
CDialog::HandleSetFont 1 0 4 4 0 100.00 %
CDialog::PreInitDialog 1 0 2 2 0 100.00 %
CDialog::OnSetFont 1 0 2 2 0 100.00 %
CDialog::OnCancel 1 0 3 3 0 100.00 %
32473 2895 29578 AVG=9.70 %
Editor
Source Line | Hit Count | Mark
File Name: c:\program files\microsoft visual studio\vc98\mfc\src\dlgdata.cpp
HWND CDataExchange::PrepareCtrl(int nIDC)
// Hit Count : 3
// Total Lines : 13
// Lines Uncovered : 6
// % Covered : 53.85 %
{
ASSERT(nIDC != 0);
ASSERT(nIDC != -1); // not allowed
HWND hWndCtrl;
35 3
36 6
37 6
35:1 Total lines: 669; File size: 18,690 bytes
Event View Disassembler Editor Details Call Graph Call Tree

The column footer shows summary results for the values displayed in that column. You can customize the summary type and summary format using the **Format Columns** dialog. For instance, you can select one of the five summary types (Sum, Count, Avg, Min, Max) or you can hide the summary for the column.

Note that by default the Report panel shows only a shred of available columns. You can easily add more columns to the panel. For more information on this, see *Adding and Removing Columns*. You can arrange the columns in the panel as you desire: move columns, change column width, etc. For more information on this, see *Arranging Columns, Lines and Panels*.

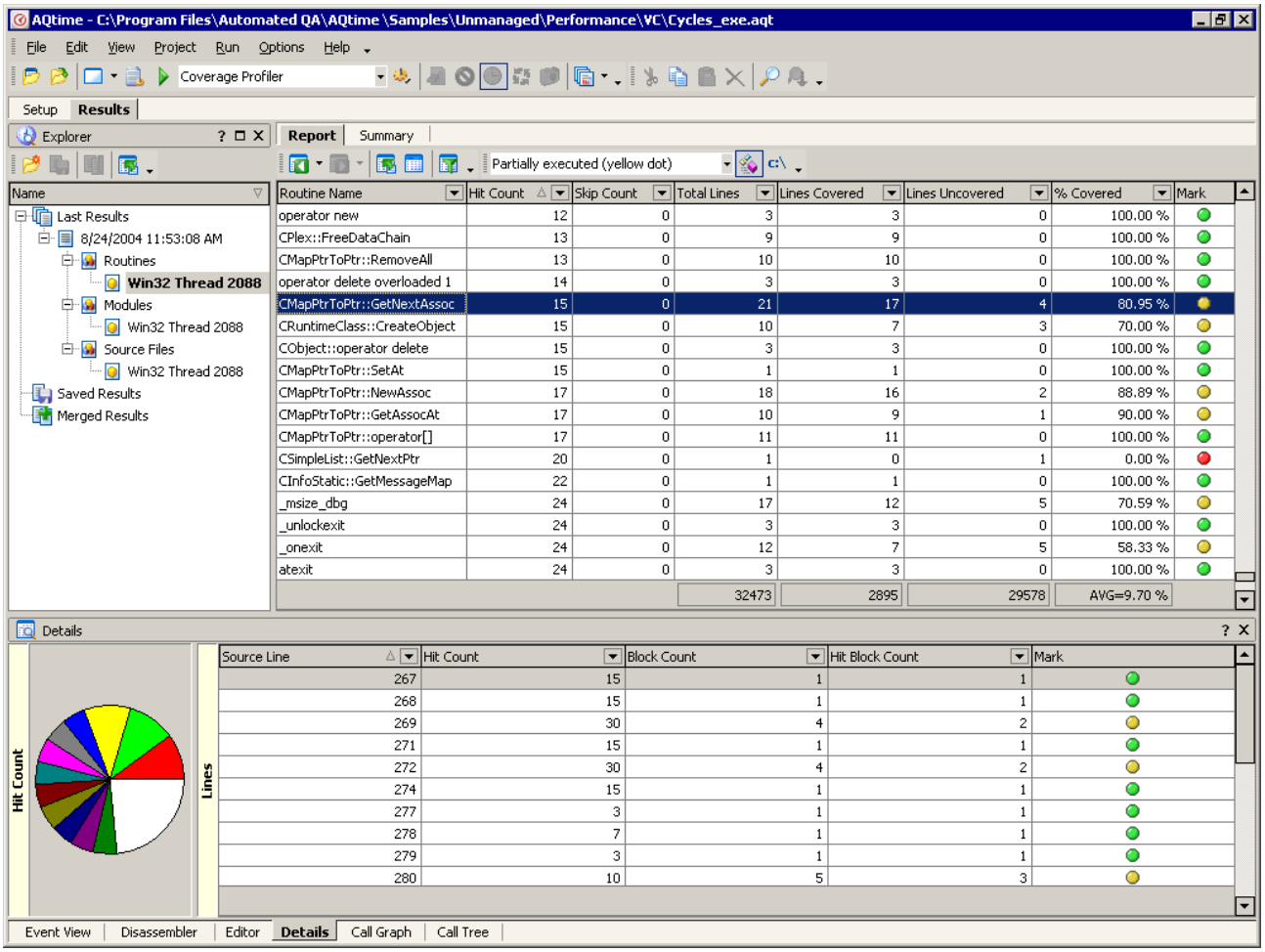
The **Profiler** toolbar contains items that allow you to modify the results that are currently being displayed as your needs dictate. For example, the *Show non-hit routines* toolbar item lets you easily include or exclude non-executed routines from the result display. For more information on the toolbar items, see *Coverage Profiler Options*.

Coverage Profiler Results - Details and Editor Panels

The **Details** and **Editor** panels display profiling results if you select the *Routines* category. When this category is active, each row in the Report panel corresponds to a single routine in your application.

Double-clicking on a routine in the Report panel will update the contents of the Details and Editor panels so they will display information concerning that routine.

The **Lines** page of the Details panel holds line coverage profiling results. Each row in this page corresponds to a source code line. You can work with rows on the Lines page in the same manner as you work with rows in the Report panel. For more information on the Details panel columns, see *Coverage Profiler - Details Panel Columns*. The chart on the left of the Details grid graphically illustrates profiling results:


AQtime - C:\Program Files\Automated QA\AQtime \Samples\Unmanaged\Performance\VC\Cycles_exe.aqt
File Edit View Project Run Options Help
Coverage Profiler
Setup Results
Explorer
Report Summary
Partially executed (yellow dot)
Name
Last Results
8/24/2004 11:53:08 AM
Routines
Win32 Thread 2088
Modules
Win32 Thread 2088
Source Files
Win32 Thread 2088
Saved Results
Merged Results
Routine Name | Hit Count | Skip Count | Total Lines | Lines Covered | Lines Uncovered | % Covered | Mark
operator new 12 0 3 3 0 100.00 %
CPlex::FreeDataChain 13 0 9 9 0 100.00 %
CMapPtrToPtr::RemoveAll 13 0 10 10 0 100.00 %
operator delete overloaded 1 14 0 3 3 0 100.00 %
CMapPtrToPtr::GetNextAssoc 15 0 21 17 4 80.95 %
CRuntimeClass::CreateObject 15 0 10 7 3 70.00 %
CObject::operator delete 15 0 3 3 0 100.00 %
CMapPtrToPtr::SetAt 15 0 1 1 0 100.00 %
CMapPtrToPtr::NewAssoc 17 0 18 16 2 88.89 %
CMapPtrToPtr::GetAssocAt 17 0 10 9 1 90.00 %
CMapPtrToPtr::operator[] 17 0 11 11 0 100.00 %
CSimpleList::GetNextPtr 20 0 1 0 1 0.00 %
CInfoStatic::GetMessageMap 22 0 1 1 0 100.00 %
_msize_dbg 24 0 17 12 5 70.59 %
_unlockexit 24 0 3 3 0 100.00 %
_onexit 24 0 12 7 5 58.33 %
atexit 24 0 3 3 0 100.00 %
32473 2895 29578 AVG=9.70 %
Details
Hit Count
Lines
Source Line | Hit Count | Block Count | Hit Block Count | Mark
267 15 1 1
268 15 1 1
269 30 4 2
271 15 1 1
272 30 4 2
274 15 1 1
277 3 1 1
278 7 1 1
279 3 1 1
280 10 5 3
Event View Disassembler Editor Details Call Graph Call Tree

Note: Compilers can divide some source lines into several blocks. This typically occurs with the `if…then` statements. Suppose, you have the following code:

```
if a < b or c > d then DoSomething;
```

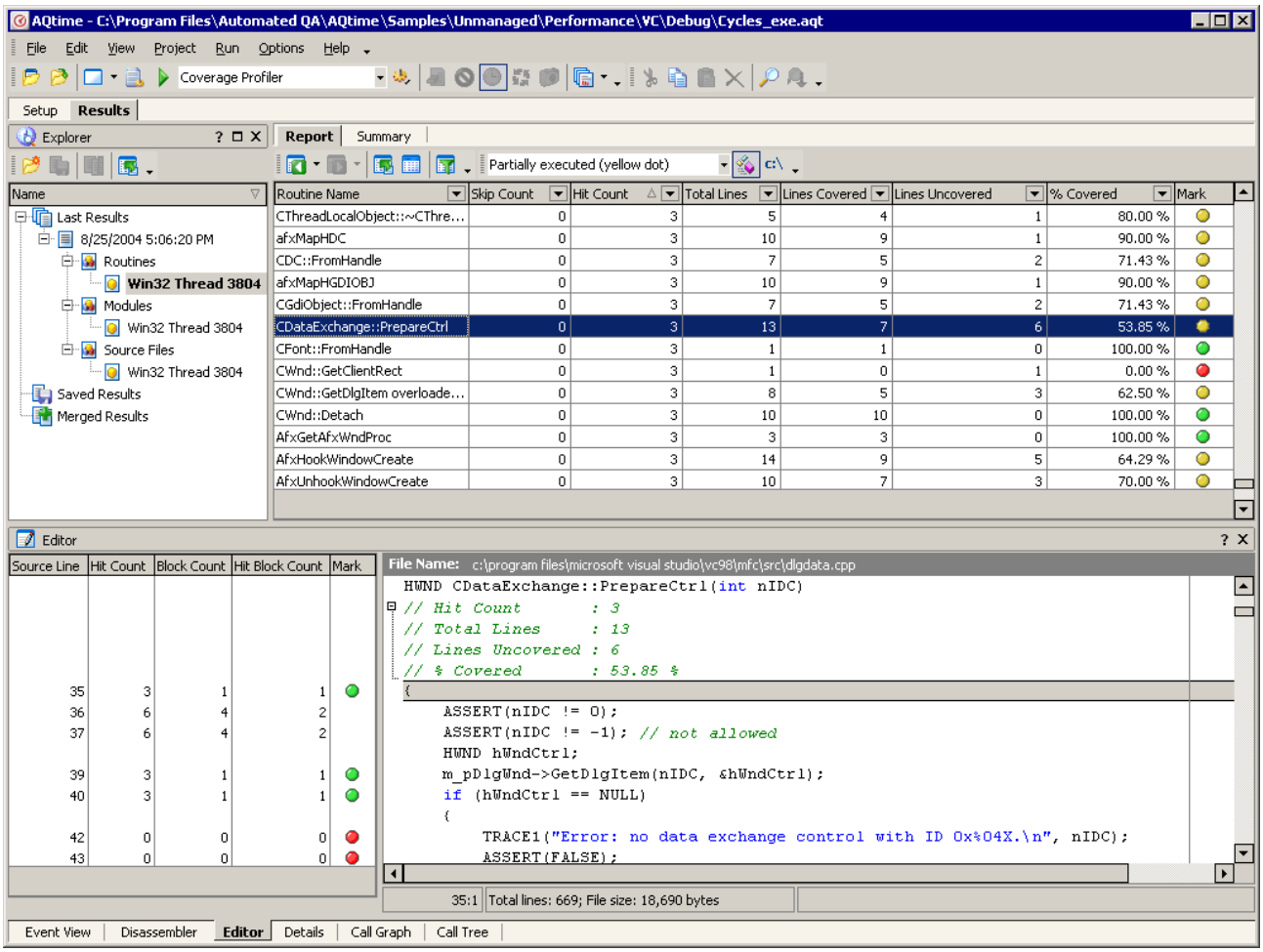
This line can be divided into three blocks: 1) the `a < b` condition, 2) the `c > d` condition and 3) call to `DoSomething`. Such lines are called ***partially executed lines***. Even if the `DoSomething` routine was called, it is not easy to say which of the conditions were executed. Another example of an instruction that is divided by the compiler into several blocks, and which is rather frequent, is Delphi's `div`. It performs integer division. Delphi's compiler breaks this instruction into several blocks when the divisor is a power of 2.

The **Block Count** column in the Details panel shows the number of block into which a source code line is divided.

The Coverage profiler includes the *Mark partially executed lines as* option that specifies how AQtime treats partially executed lines when it's calculating the Lines Covered, Lines Uncovered and % Covered values and how AQtime marks such lines (with a red, green or a yellow dot) in the Lines page of the Details panel and in the Editor (see below). This option can have one of the three following values:

Value	Description
Partially executed	AQtime marks partially executed lines with yellow dots and treats them as unexecuted when calculating the Lines Covered and Lines Uncovered columns.
Completely executed	AQtime marks partially executed lines with green dots and treats them as executed.
Non-executed	AQtime marks partially executed lines with red dots and treats them as unexecuted.

Double-clicking a row in the Details grid will move the cursor in the **Editor** panel to the source code line for which that row displays results. If a routine was profiled at line level, the Editor's grid will show the same results as the ones shown in the Details panel. For instance, you will find that the executed function and lines are marked with green dots and the unexecuted routines and line have red dots. Partially executed routines and lines are marked with yellow dots. To select which columns to display in the gutter, use the Field Chooser window. To bring it up, select **Field Chooser** from the context menu. See *Adding and Removing Columns*.


AQtime - C:\Program Files\Automated QA\AQtime\Samples\Unmanaged\Performance\VC\Debug\Cycles_exe.aqt
File Edit View Project Run Options Help
Coverage Profiler
Setup Results
Explorer
Report Summary
Partially executed (yellow dot)
Name
Last Results
8/25/2004 5:06:20 PM
Routines
Win32 Thread 3804
Modules
Win32 Thread 3804
Source Files
Win32 Thread 3804
Saved Results
Merged Results
Routine Name | Skip Count | Hit Count | Total Lines | Lines Covered | Lines Uncovered | % Covered | Mark
CThreadLocalObject::~CThre... 0 3 5 4 1 80.00 %
afxMapHDC 0 3 10 9 1 90.00 %
CDC::FromHandle 0 3 7 5 2 71.43 %
afxMapHGDIOBJ 0 3 10 9 1 90.00 %
CGdiObject::FromHandle 0 3 7 5 2 71.43 %
CDataExchange::PrepareCtrl 0 3 13 7 6 53.85 %
CFont::FromHandle 0 3 1 1 0 100.00 %
CWnd::GetClientRect 0 3 1 0 1 0.00 %
CWnd::GetDlgItem overloade... 0 3 8 5 3 62.50 %
CWnd::Detach 0 3 10 10 0 100.00 %
AfxGetAfxWndProc 0 3 3 3 0 100.00 %
AfxHookWindowCreate 0 3 14 9 5 64.29 %
AfxUnhookWindowCreate 0 3 10 7 3 70.00 %
Editor
Source Line | Hit Count | Block Count | Hit Block Count | Mark
File Name: c:\program files\microsoft visual studio\vc98\mfc\src\dlgdata.cpp
HWND CDataExchange::PrepareCtrl(int nIDC)
// Hit Count : 3
// Total Lines : 13
// Lines Uncovered : 6
// % Covered : 53.85 %
{
ASSERT(nIDC != 0);
ASSERT(nIDC != -1); // not allowed
HWND hWndCtrl;
m_pDlgWnd->GetDlgItem(nIDC, &hWndCtrl);
if (hWndCtrl == NULL)
{
TRACE1("Error: no data exchange control with ID 0x%04X.\n", nIDC);
ASSERT(FALSE);
35:1 Total lines: 669; File size: 18,690 bytes
Event View Disassembler Editor Details Call Graph Call Tree

Note for users who work in Visual Studio: The Code Editor of Visual Studio lets you collapse and expand blocks of source code. The grid, which AQtime adds to the Code Editor to display profiling results,

supports neither collapsing, nor expanding, because Visual Studio does not send appropriate notifications to AQtime. So, to ensure that the grid shows proper profiling results for source lines and routines, please expand all the collapsed blocks of code. To do this, use the **Outlining | Toggle All Outlining** or **Outlining | Stop Outlining** item of the Code Editor's context menu.

Note for users who work in Borland Developer Studio: The Editor of Borland Developer Studio lets you collapse and expand blocks of source code. The grid, which AQtime adds to the Editor to display profiling results, supports neither collapsing, nor expanding, because Borland Developer Studio does not send appropriate notifications to AQtime. So, to ensure that the grid shows proper profiling results for source lines and routines, please expand all the collapsed blocks of code. To do this, use the **Unfold | All** item of the Editor's context menu.

Some notes on displaying results in the Editor panel:

- Note that in order for AQtime to show source files in the Editor, the path to these files must be specified in the Project Search Directories or Search Directories dialogs. In addition, your applications must be compiled with debug information (see *How AQtime Profilers Use Metadata and Debug Information*).
- Information about lines depends on debug info attached to the executable. With the Coverage profiler especially, you should be on the lookout for unexpected discrepancies. Some compilers, for instance, such as Borland Delphi, will skip functions that are never called (this is called Smart Linking). Therefore, the debug information will log fewer functions and fewer lines than there are in the source file.
- The Editor can display incorrect profiling-related information for some C++ applications that use several functions based on the same template (see *Profiling Template Functions*). In this case, refer to the Report panel to get the correct results.
- Sometimes the Coverage profiler can report that the last line of your routine was not executed (it shows a red dot for this line in the Editor's grid). The reason for this is that AQtime stops profiling a routine when the application executes the `ret` instruction. Usually, this instruction is the last instruction in the routine's binary code. However, compilers can produce code which includes `ret` instructions in "the middle"of a routine. For example, the following C++Builder code will insert `ret` instructions after each `case` line of a `switch... case` block.

  [C++]

```
void foo(int i)
{
  . . .
  switch(i)
  {
  case 1: //do something
    break;
  case 2: // do something
    break;
  }
} // This line is never executed
```

Coverage Profiler - Report Panel Columns

When displaying results of the **Coverage** profiler, each row in the **Report** panel holds the results for a routine, source file or module in your application. Which values are displayed depends on the category selected in the **Explorer** panel.

Routines Category

Columns (in alphabetical order)	**Description**
Address	Routine's address in memory. This column is used for unmanaged (native-code) routines only.
Analysis Result	Specifies if the routine was instrumented or not. If the routine was instrumented, this column is empty. Otherwise, the column displays a short description why the routine was not instrumented: • *Less than 5 bytes* - The routine occupies less than 5 bytes in memory. See *Profiling Small Functions*. • *No line info* - The routine was added to a line-level area, but the debug information holds no info about routine lines. These routines can be profiled at routine level only. • *Unsafe code* - AQtime was not able to instrument the routine safely. This typically occurs when the routine's binary code is intermixed with data areas. See *Profiling Routines That Hold Unsafe Code*. • *No ret instruction* - The routine's binary code does not contain the ret instruction (this may happen if the routine finishes with the jmp instruction). See *Profiling Routines That Do Not Have the ret Instruction*. • *Duplicated code* - The routine whose code coincides with code of another routine. To learn more about this, see *Profiling Duplicated Code*.
Class Name	If the routine is a method, name of the class it belongs to.
Code Type	Specifies the type of the routine's code. The following values are possible: • **MSIL** - Managed-code routine with MSIL (Microsoft Intermediate Language) code. • **x86** - Native-code (unmanaged) routine. • **Pseudo** - Pseudo routine that was created by the context. For example, *<JIT Compiler>*, *<Garbage Collector>*, *<Unknown PInvoke>* or *<Root>*. • **PInvoke** - Native-code routine for which there is a declaration in one of managed modules and that is called from within the unmanaged code. • **NGen** - Managed routine that was compiled by the *ngen* utility (CLR Native Image Generator) with the */prof* argument in its command line. The *ngen* compilation means the routine was compiled before the application starts. • **Script** - The routine belongs to a script that was profiled along

	with the host application. See *Profiling Scripts* for details.
Hit Count	The number of routine calls that were profiled. See also *Skip Count*. The total number of times the routine was executed is determined as *Hit Count + Skip Count*.
Lines Covered	Specifies the number of routine's source lines that were executed during the profiler run.
Lines Uncovered	Specifies the number of routine's source lines that were not executed during the profiler run. **Note:** According to the *Mark partially executed lines as* option, partially executed lines can be treated either as executed or as non-executed. If the routine was profiled at routine level, AQtime considers all the lines of this routine as executed (if the routine was called) or unexecuted (if the routine was not called).
Mark	Specifies if the routine was executed or not during the profiler run. If the routine was profiled at *routine level*, Mark holds green dot if the method was executed or red dot if the method was not executed. If the method was profiled at *line level*, Mark holds green dot if all lines of the method were executed, red dot if no line of the method was executed, yellow dot if some lines were executed and some were not.
Module Name	The name of the module which contains the profiled routine.
Namespace	Namespace of the method's class (this column is used for managed routines only).
Routine Name	Name of the routine.
Skip Count	Number of times the routine was excluded from profiling, because the profiling status was off (this can be, for example, the number of times the routine was affected by an off-trigger or the number of times the routine was executed when the Enable/Disable Profiling button was not pressed). See also *Hit Count*. The total number of times the routine was executed is determined as *Hit Count + Skip Count*.
Source File	Name of the source file for the method. The values for this column are read from the application's debug info.
Source Line	Source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.
Token	The routine's token. This column is used for managed routines only.
Total Lines	The total number of source code lines in the routine.
Unit Name	Name of the linkage unit that holds the routine. This column is used for unmanaged (native-code) routines only.
% Covered	The percentage of covered lines against the total number of lines in the routine. Partially executed lines are counted according to the *Mark partially executed lines as* option value.

Source Files and Modules Categories

Column (in alphabetical order)	**Description**
File Name	Name of the source file (or module).

or **Module Name**	
Hit Count	The number of routine calls that were profiled. See also *Skip Count*. This value is a sum of the Hit Count result of all profiled routines that belong to the given source file or module.
Skip Count	Number of times the routines were excluded from profiling, because the profiling status was off (This can be, for example, the number of times the routines were affected by off-triggers). This value is a sum of the Skip Count result of all profiled routines that belong to the given source file or module.
% Covered	The percentage of covered lines against the total number of lines in the routines that belong to the source file (module). Partially executed lines are counted according to the *Mark partially executed lines as* option value.

Note that by default the Coverage profiler shows a few of available columns in the Report panel. You can add more columns to the panel. For more information, see *Adding and Removing Columns*.

Coverage Profiler - Details Panel Columns

If a routine was profiled at routine level, the **Coverage** profiler uses the **Details** panel to display additional profiling results for that routine. The panel holds the **Lines** table that shows the line profiling results for the routine selected in the Report panel. This table is populated only if the routine was profiled at line level. Each row in the table holds profiling results for a source code line:

Column (in alphabetical order)	**Description**
Block Count	Specifies the number of blocks into which the source code line is divided. For more information, see description of partially executed lines.
Hit Block Count	Specifies the number of the line's blocks that were called during the profiler run.
Hit Count	Specifies how many times the source line was executed during the profiler run. Note that for lines that constitute several blocks in the compiled code (that is, for lines whose *Block Count* is more than 1), this value is the total hit count for all the blocks corresponding to the line. For more information, see the description of partially executed lines in the *Coverage Profiler – Overview* topic.
Mark	Specifies whether the source line was executed or not. If the line was executed, the Mark column holds a green dot. If the line was not executed, Mark holds a red dot. Partially executed lines are marked with yellow dots. Note that according to the *Mark partially executed lines as* option, partially executed lines can be treated either as executed or as non-executed. If the routine was profiled at routine level, AQtime considers all the lines of this routine as executed (if the routine was called) or unexecuted (if the routine was not called).
Source Line	The line number in the source file.

Coverage Profiler Options

The **Coverage** profiler includes two groups of customizable options:

- One group contains options that affect the current result display. When you change these options, AQtime refreshes the data in its panels.
- Another group includes options that have effect on the profiler functioning. Changes in these options will only apply to future profiler runs.

To modify options that have effect on the way the profiler functions, do any of the following:

- *AQtime standalone:*
  - select **Options | Options** from the main menu and then choose **Profilers | Coverage | Coverage Profiler** on the left of the ensuing Options dialog;
  - press **Configure Current Profiler** on the **Standard** toolbar when the **Coverage** profiler is selected.
- *AQtime integrated into Visual Studio:*
  - select **Tools | Options** from Visual Studio's main menu and then choose **AQtime | Profilers | Coverage | Coverage Profiler** from the tree view on the left of the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - select **Profile | Options** from Borland Developer Studio's main menu and then choose **Profilers | Coverage | Coverage Profiler** on the left of the ensuing Options dialog.

This group includes the following options:

- **Disable inlining** - This option effects managed (.NET) applications only. Inlining typically increases the speed and reduces the number of separate JITting events for inlined methods. However, if a method is inlined, AQtime is unable to collect the coverage information for this method and its lines. If the option is enabled, inlining of managed routines is disabled, so AQtime can profile them.
- **Thread model** - Specifies how the Coverage profiler gathers statistics for threads in the profiled application. For more information on available values for this option, see *Profiling Multiple Threads*.

To modify options that affect the way results are displayed, use items of the **Profiler** toolbar (by default, it is displayed at the top of the Report panel). If this toolbar is hidden, right-click somewhere in the toolbar area and select **Profiler** from the subsequent popup list. On the toolbar, the following items are available:

- **Mark partially executed lines as** – Specifies how AQtime treats partially executed lines when it is calculating the number of covered and un-covered lines and how AQtime marks them (with a red, green or a yellow dot) in the **Lines** pane of the **Details** panel and in the **Editor**. This option can have one of three values:

Value	Description
Partially executed	AQtime marks partially executed lines with yellow dots and treats them as non-executed when calculating the **Lines Covered**, **Lines Uncovered** and **% Covered** values.
Completely executed	AQtime marks partially executed lines with green dots and treats them as executed when calculating the **Lines Covered**, **Lines Uncovered** and **% Covered** values.
Non-executed	AQtime marks partially executed lines with red dots and treats them as non-executed.

- **Show routines with class names** – If it is enabled, the **Routine Name** column of the **Report** panel for the Coverage profiler displays the name of the given routine along with the name of the class the routine belongs to. Otherwise, this column only displays the routine name.
- **File names with path** – If this option is enabled, the **Source File** and **Module Name** columns of the Report panel for the Coverage profiler hold the entire path to the given file. Else, these columns hold the file name only.

Exception Trace Profiler

The **Exception Trace** profiler monitors the application execution and if an exception occurs it outputs exception information (exception type, address, call stack, and so forth) on the Event View panel. If you want the call stack to be reported to a file, enable the panel's *Text file* | *Active* or *XML file* | *Active* option. You can also use panel settings to filter logged exceptions (see *Exceptions in the Event View Panel*).

The Exception Tracer supports both Win32 and Win64 applications, works faster in comparison with other profilers and does not slow down the tested application. Use it if you only need to explore exceptions that occur during the application execution.

You can view the source code of a routine in the call stack: simply double-click the routine in the Event View panel and then switch to the **Editor** (Note that the path to the source files must be specified in the Project Search Directories or Search Directories dialogs).

Since the Exception Tracer uses the Event View panel, the *Exceptions* | *Active option* of this panel must be turned on. Otherwise, the Exception Tracer displays an error message and does not start profiling. The other Event View options specify what information will be displayed when the exceptions occur.

Function Trace Profiler

Function Trace Profiler – Overview

The **Function Trace** profiler lets you investigate the route and the order in which the routines are called during the application runtime. It traces both 32- and 64-bit Windows and .NET applications and gathers information about each application function: different routes used to invoke the function, exact parameter values passed to the function, hierarchy of function calls, and so on. The complete profiler description includes the following topics:

- *Overview (this topic)*
- *Tracing Function Call Parameters and Result Values*
- *Description of the Report Panel Columns*
- *Description of the Details Panel Columns*
- *Outputting Results Using a Custom DLL*
- *Function Trace Profiler Options*

Overview

The **Function Trace** profiler provides you with comprehensive information about routine usage. It is a good way to find out how the routine can be called or to know the actual call stack for a function (for instance, a function that raises an exception). Also, you can use it when you need to know whether something occurs in

the application at the expected point, for instance, whether the application posts data to the database right after a user has pressed OK. Another good use of the profiler, again, is to profile an application with complex recursive calls.

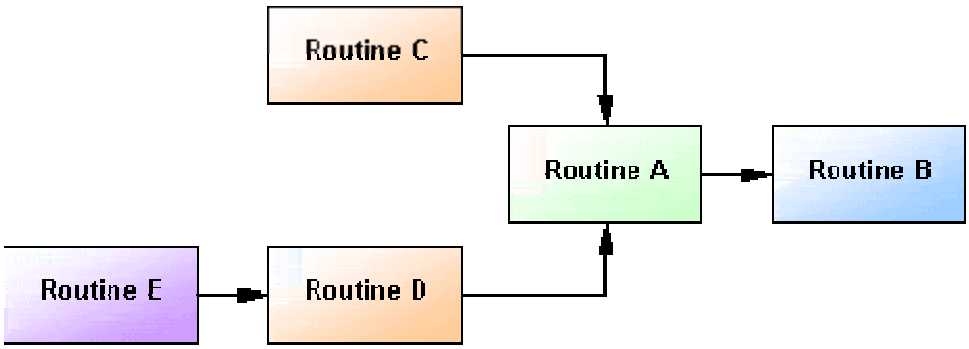
After you start the Function Trace profiler, it monitors the application's execution flow, logs call stacks for each call of a routine and places gathered results into **Routines** and **Call Trace** categories.

The **Routines** category is used to display route information. Note that route tracing consumes a lot of memory and by default this functionality is turned off. To enable it, specify any value that is greater than 0 for the profiler's *Maximum route depth* option. You can set this value in the Run Settings dialog that is displayed every time you start profiling. If the option is 0, the profiler does not trace the routes.

If route tracing was enabled, when you select the **Routines** category in the Explorer panel, the Report panel lists all routines used by the application. The columns of the Report panel contain detailed information about each routine (see *Function Trace - Report Panel Columns*).

When the Routines category is chosen the **Report** panel lists all routines used by the application. The columns of the Report panel contain detailed information on each routine (see *Function Trace Profiler - Report Panel Columns*).

It is obvious that a routine can be called by several different routines in your application. For instance, for routine B displayed on the figure below, there are two calling routes: C -> A -> B and E -> D -> A -> B.



With the Function Trace profiler you can find all routes used to call the selected routine. Note that this is not the same as building the calls relationship (parents-children) graph. The calls relationships graph includes all "ancestors" of a routine without specifying the exact calling routes. For instance, in the Call Graph, generated for routine B, it would be rather difficult to find whether the route E -> D -> A -> B exists or if it is just a combination of two routes: D -> A -> B and E -> D.

To view the call routes for a routine, choose the **Routines** category, select the routine in the Report panel and switch to Details. The **Details** panel includes two panes: Call Routes and Call Stack. Each call to a function has its call stack (or call route). The Call Routes pane holds a list of all call routes for the selected function. The Route No column specifies the route number (AQtime enumerates routes in order of their "appearance"), the HitCount column specifies how many times the function was executed with this call route. The call route itself is displayed in the Call Stack pane. The first row in this pane displays the selected routine itself, the second row – direct parent of the selected routine, the third row - the function that called the direct parent, and so on (see *Description of the Details Panel Columns*).

Note: If the *Maximum route depth* option is 0 before the profiler starts, the profiler does not trace call routes, so the Details panel is empty.

When the **Call Trace** category is active the Report panel displays the sequence of routine calls and call characteristics. Which call characteristics to measure is defined by the *Active counter* option. The following counters are available in the current AQtime version (see *Counters - Overview* for counter descriptions):

- *Elapsed Time*
- *User Time*
- *User+Kernel Time*
- *CPU Mispredicted Branches*
- *CPU Cache Misses*
- *Context Switches*
- *64K Aliasing Conflicts*
- *Split Load Replays*
- *Split Store Replays*
- *Blocked Store Forwards Replays*
- *Soft Memory Page Faults*
- *Hard Memory Page Faults*
- *All Memory Page Faults*

All the counters work both for managed and unmanaged code and support both 32bit and 64bit applications. For a complete description of counters, see *Counters Overview*.

Some counters may be unavailable. This depends on the CPU model and the software used. For instance, some counters do not work on Pentium II or do not support the processor's SpeedStep technology, while others do not function under virtual machines. Also, if you run AQtime x86 on a 64-bit operating system, the only available counter is *Elapsed Time*. For complete information on known counter restrictions, see *Counters Overview*.

Also, if you have Windows DDK installed, using some counters may cause the operating system to stop unexpectedly and display the error description on a blue screen. For more information on this problem and on how to solve it, see *Counters - Overview*.

Note: If you use a computer that has several processors or a multiple-core processor (for example, dual-core CPU) and has Windows XP Service Pack 2, then you must install the Windows update #896256 in order for the profiler to be able to time your application correctly. The update is available on Microsoft's web site:

http://support.microsoft.com/kb/896256

The hierarchy of calls and some measured characteristics for the routine selected in the Report panel are displayed in the **Call Tree** and **Call Graph** panels.

One of the Function Trace benefits is that the profiler does not just log the sequence of function calls, but also logs the parameters of function calls and function result values. For complete information on this, see *Function Trace Profiler - Tracing Function Call Parameters and Result Values*.

Besides generating results at the end of profiling or on demand (with the **Get Results** command), the Function Trace profiler can output real-time information into a text file, custom DLL or CodeSite debugging tool:

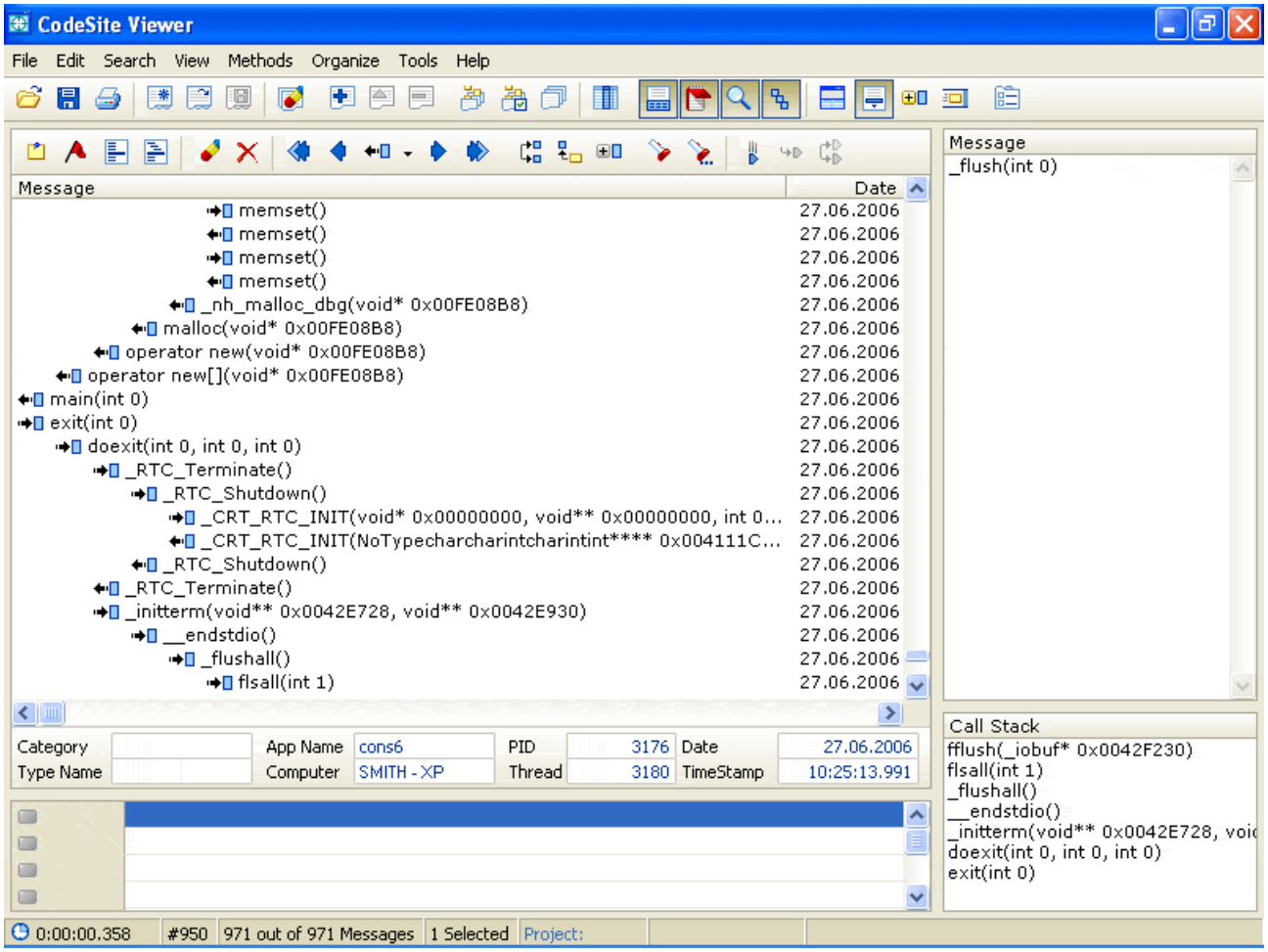
- When the *Text file output* option is enabled the routine call order is written to one or several text files.

  In the folder specified by the *Text output directory* option a new folder is created. The folder name corresponds to current date and time (the date is represented in the format specified by Windows Regional Settings, the time is repesented in the hh:mm:ss format). For each thread a separate text file is assigned where its routine sequence is written. The file name matches the thread name or number. Text files are created only for those threads from which the profiled routines are called.

  For each call, two messages are posted, one on entry (marked with "->") and one on exit (marked with "<-"). If a lot of child calls occur, the two lines may be far apart. If the routine belongs to an area whose *Retrieve parameter values* property is enabled, the posted text also includes the parameter names and values.

- When the *External DLL output* option is enabled a custom dynamic link library is used to output profiler results. With a custom DLL you have complete control on how the profiler results are processed. For more information on this, see *Outputting Results Using a Custom DLL*.
- When the *CodeSite 1 output* or *CodeSite 2/3 output* option is enabled CodeSite versions 1, 2 or 3 are used to output routine calls.

  CodeSite is the product of Raize Software (www.raize.com), it is not supplied with AQtime.



Tracing function calls with the CodeSite3 Viewer

If CodeSite is not running when you start the Function Trace profiler, the profiler will automatically launch it. If CodeSite is already running, Function Trace will use the running instance of the application to output results. The profiler does not close CodeSite when the profiling run is over, thus making it possible for you to analyze profiling results.

The Function Trace profiler shows each call, with the amount of detail you specify in the Options. It is not a statistical tool, but very detail-oriented. It is very easy to slow your application to a crawl and to generate a flood of detail, simply by letting the Function Trace profile too much of the application in one run. Use Areas, Triggers, and Files to Ignore or Routines to Ignore dialogs to limit the profiler usage. See also *Controlling What to Profile* and *Excluding Code From Profiling* topics.

Function Trace Profiler – Tracing Function Call Parameters and Result Values

The **Function Trace** profiler can log information about function call parameters and return values. This information will be logged if the routine belongs to an including area whose *Retrieve parameter values* property is enabled. This option is available when creating a new area or when modifying the properties of an existing area.

After you perform the profiling and get results, activate the *Call Trace* results category in the **Explorer** panel. When this category is chosen, the **Report** panel displays a sequence of function calls for the application thread that is selected in the Explorer panel. The **Details** panel contains parameter information for the routine that is selected in the Report panel. The Details panel displays information on function parameters in two tables: **Routine Parameters On Enter** and **Routine Parameters On Exit**. These tables show parameter values for entering and exiting the routine. The last row in the Routine Parameters On Exit table holds information about the routine results. For more information on panel columns, see *Function Trace Profiler - Details Panel Columns (Call Trace Categogry)*. The type of information displayed depends on several conditions:

- For some Borland VCL constructors and destructors an additional parameter can be displayed. It does not exist in the source code but is added by the compiler.
- If the application was compiled with Optimization enabled, parameter values may be incorrect. We recommend that you turn Optimization off if you want to trace parameter values on calls.
- For parameters passed by reference, the pointer address is analyzed. If TestComplete managed to trace the parameter value then the pointer address (or several addresses in a pointer-to-pointer case) is shown together with the result value, otherwise only the pointer address. The analysis result value depends on the data type. For native applications the following information is displayed:

Data type	Displayed value
Character	The ordinal number of the character.
Pointer to character	The character trailed with ellipses.
Boolean	*True* or *False* values.
Other simple data types: integer, floating point and so on	The actual parameter value.
String	Either the whole string text or the first character of the string trailed with ellipses.
Array	The first element of the array (for unmanaged routines) or the array's type (for managed routines).
Other complex data types	Only the memory address.

For parameters of .NET applications the string value returned by the `ToString()` function will be shown. For example, the following string value corresponds to an array type: *System.Int32[]*.

- For the function passed as routine parameter the following information is displayed:

Column	Value	Example
Param Type	Function result type and types of function parameters: *Result (Param1, Param2, ...)*	Integer (Integer, Boolean)

Param Value	Function address and possibly a pointer analysis result.	0x0045C8F0

- Depending on the compiler and user settings, Variant type parameters can be displayed in three ways: as address, structure or variant. In Visual Studio applications, variants are interpreted as address or structure. In Delphi applications - as address or variant. In C++Builder applications, all three interpretations are possible.
- For class instance passed by reference, only the address is shown. If the object is passed by value, it is parsed into class fields and field-values are displayed in braces. For each class field the following information is shown: type, name and value. Fields are delimited with commas. The maximum number of class fields to be displayed is specified by the *Maximum number of fields* option. If there are more fields in the class then the value list ends with ellipsis. For example, `{string Font_Name = Arial, int Font_Size = 14, ...}`.
- All of the above information also applies to routine results. The results are displayed in the **Routine Parameters On Exit** pane. The function result is distinguished with the *Return Value* mark in the Comments column. The output parameters of a procedure are not specifically marked, but you can track them by comparing the values in the **Routine Parameters On Enter** and **Routine Parameters On Exit** panes.
- Currently, the Function Trace profiler does not collect information on values of script routines' parameters.

Function Trace Profiler – Report Panel Columns

When displaying results of the Function Trace profiler, each row in the Report panel holds gathered routine results. Which values are displayed depends on the category selected in the Explorer panel and on the counter that was used for profiling.

Note that by default the Function Trace profiler shows available columns in the Report panel. You can add more columns to the panel. For more information on this, see *Adding and Removing Columns* in AQtime help.

Routines Category

When the Routines category is active, the Report panel holds descriptions for each routine that was called during profiling. The panel contains the following columns:

Columns (in alphabetical order)	**Description**
Address	Routine's address in memory. This column is used for unmanaged (native-code) routines only.
Analysis Result	Specifies if the routine was instrumented or not. If the routine was instrumented, this column is empty. Otherwise, the column displays a short description why the routine was not instrumented: *Less than 5 bytes* - The routine occupies less than 5 bytes in memory. See *Profiling Small Functions.* *No line info* - The routine was added to a line-level area, but the debug information does not hold info about routine lines. These routines can be profiled at routine level only. *Unsafe code* - AQtime was not able to instrument the routine safely. This typically occurs when the routine's binary code is intermixed with data areas. See *Profiling Routines That Hold Unsafe Code.*

	No ret instruction - The routine's binary code does not contain the `ret` instruction (this may happen if the routine finishes with the `jmp` instruction). See *Profiling Routines That Do Not Have the ret Instruction*. *Duplicated code* - The routine whose code coincides with code of another routine. To learn more about this, see *Profiling Duplicated Code*.
Class Name	If the routine is a method, name of the class it belongs to.
Code Type	Specifies the routine's code type. The following values are possible: • **MSIL** - Managed-code routine with MSIL (Microsoft Intermediate Language) code. • **x64** - 64-bit code routine. • **x86** - Native-code (unmanaged) routine. • **Pseudo** - Pseudo routine that was created by the context. For example, *<JIT Compiler>*, *<Garbage Collector>*, *<Unknown PInvoke>* or *<Root>*. • **PInvoke** - Native-code routine for which there is a declaration in one of the managed modules and that is called from within unmanaged code. • **NGen** - Managed routine that was compiled by the *ngen* utility (CLR Native Image Generator) with the */prof* argument in its command line. The *ngen* compilation means the routine was compiled before the application starts. • **Script** - The routine belongs to a script that was profiled along with the host application. See *Profiling Scripts* for details.
Hit Count	The number of routine calls that were profiled.
Module Name	The name of the module which contains the profiled routine.
Namespace	Namespace of the method's class (this column is used for managed routines only).
Routine Name	Name of the routine.
Source File	Name of the source file for the method. The values for this column are read from the application's debug info.
Source Line	Source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.
Token	The routine's token. This column is used for managed routines only.
Unit Name	Name of the linkage unit that holds the routine. This column is used for unmanaged (native-code) routines only.

Information for all routes that were used to invoke a selected routine is shown in the **Details** panel. See *Function Trace Profiler - Details Panel - Routines Category* for more information.

Call Trace Category

When the Call Trace category is active, the **Report** panel holds results for each routine call. Which values are displayed also depends on the counter that was used for profiling.

Columns that do not depend on the active counter

Columns (in alphabetical order)	**Description**
Analysis Result	Specifies if the routine was instrumented or not. If the routine was instrumented, this column is empty. Otherwise, the column displays a short description why the routine was not instrumented: *Less than 5 bytes* - The routine occupies less than 5 bytes in memory. See *Profiling Small Functions.* *No line info* - The routine was added to a line-level area, but the debug information does not hold info about routine lines. These routines can be profiled at routine level only. *Unsafe code* - AQtime was not able to instrument the routine safely. This typically occurs when the routine's binary code is intermixed with data areas. See *Profiling Routines That Hold Unsafe Code*. *No ret instruction* - The routine's binary code does not contain the `ret` instruction (this may happen if the routine finishes with the `jmp` instruction). See *Profiling Routines That Do Not Have the ret Instruction*. *Duplicated code* - The routine whose code coincides with code of another routine. To learn more about this, see *Profiling Duplicated Code*.
Call No	The number of a single routine call.
Class Name	If the routine is a method, name of the class it belongs to.
Code Type	Specifies the routine's code type. The following values are possible: • **MSIL** - Managed-code routine with MSIL (Microsoft Intermediate Language) code. • **x64** - 64-bit code routine. • **x86** - Native-code (unmanaged) routine. • **Pseudo** - Pseudo routine that was created by the context. For example, *<JIT Compiler>*, *<Garbage Collector>*, *<Unknown PInvoke>* or *<Root>*. • **PInvoke** - Native-code routine for which there is a declaration in one of the managed modules and that is called from within unmanaged code. • **NGen** - Managed routine that was compiled by the *ngen* utility (CLR Native Image Generator) with the */prof* argument in its command line. The *ngen* compilation means the routine was compiled before the application starts. • **Script** - The routine belongs to a script that was profiled along with the host application. See *Profiling Scripts* for details.
Module Name	The name of the module which contains the profiled routine.
Parent Name	Name of the routine that called the given routine.
Routine Name	Name of the routine.

Source File	Name of the source file for the method. The values for this column are read from the application's debug info.
Source Line	Source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.
Unit Name	Name of the linkage unit that holds the routine. This column is used for unmanaged (native-code) routines only.

Columns specific to the *Elapsed Time*, *User Time* and *User+Kernel Time* counters

You can specify the measurement unit for the following columns (*seconds*, *milliseconds*, *microseconds* or *machine cycles*) using the **Counter unit** box on the **Profiler** toolbar. See also *Function Trace Profiler Options*.

Columns (in alphabetical order)	**Description**
Time	Total time spent executing the routine's code excluding child calls. The sum of all profiled methods appears in the footer of this column.
Time with Children	Total time spent on calls to the routine including calls to child routines. The sum for all profiled routines is displayed in the footer of this column.

Columns specific to the *CPU Cache Misses* counter

Columns (in alphabetical order)	**Description**
Misses	Total number of cache misses that occurred during execution of the routine's code excluding child calls. The sum for all profiled routines appears in the footer of this column.
Misses with Children	Total number of cache misses that occurred during execution of the routine (including its calls to child methods). The sum for all profiled routines is displayed in the footer of this column.

Columns specific to the *CPU Mispredicted Branches* counter

Columns (in alphabetical order)	**Description**
Branches	Total number of branches that were mispredicted during execution of the routine's code excluding child calls. The sum for all profiled routines appears in the footer of this column.
Branches with Children	Total number of branches that were mispredicted during execution of the routine (including mispredictions in child methods). The sum for all profiled routines is displayed in the footer of this column.

Columns specific to the *Hard Memory Page Faults*, *Soft Memory Page Faults* and *All Memory Page Faults* counters

Columns (in alphabetical order)	**Description**
Faults	Total number of page faults that occurred during execution of the routine's code (child calls are excluded). The sum for all profiled routines appears in the footer of this column.
Faults with Children	Total number of page faults that occurred during execution of the routine (including page faults that occurred in child methods). The sum for all profiled routines is displayed in the footer of this column.

Columns specific to the *Split Load Replays, Split Store Replays* and *Blocked Store Forwards Replays* counters

Columns (in alphabetical order)	**Description**
Replays	Total number of replays that occurred during execution of the routine's code (child calls are excluded). The sum for all profiled routines appears in the footer of this column.
Replays with Children	Total number of replays that occurred during execution of the routine (including replays that occurred in child methods). The sum for all profiled routines is displayed in the footer of this column.

Columns specific to the *64K Aliasing Conflicts* counter

Columns (in alphabetical order)	**Description**
Conflicts	Total number of aliasing conflicts that occurred during execution of the routine's code (child calls are excluded). The sum for all profiled routines appears in the footer of this column.
Conflicts with Children	Total number of aliasing conflicts that occurred during execution of the routine (including conflicts that occurred in child methods). The sum for all profiled routines is displayed in the footer of this column.

Columns specific to the *Context Switches* counter

Columns (in alphabetical order)	**Description**
Switches	Total number of context switches that occurred during execution of the routine's code (child calls are excluded). The sum for all profiled routines appears in the footer of this column.
Switches with Children	Total number of context switches that occurred during execution of the routine (including switches that occurred in child methods). The sum for all profiled routines is displayed in the footer of this column.

Additional information about routines chosen in the Report panel is shown in the **Details**, **Call Tree** and **Call Graph** panels. Information displayed in the Call Tree and Call Graph panels help you trace the routine call hierarchy, whereas the Detail panel displays exact parameters passed to the selected routine. See column descriptions in the *Function Trace Profiler - Details Panel Columns* topic.

Function Trace Profiler – Details Panel Columns

The **Function Trace** profiler organizes results into two categories: Routines and Call Trace. When you view the Function Trace profiler results for your routines, the contents of the **Details** panel depends on the category selected in the **Explorer** panel.

Note that the Details panel does not display all available columns. You can add columns to the panel or remove them from it as it is described in *Adding and Removing Columns* in AQtime help.

Routines Category

When the Routines category is active the **Report** panel contains a list of all profiled routines. The **Details** panel displays all routes that were used to call a selected routine. It consists of two panes: **Call Routes** and **Call Stack**. The Call Routes pane serves to select a distinct routine and Call Stack shows a sequence of routines that invoke the chosen routine.

© 2010 AutomatedQA Corp.

Note: The Function Trace profiler only traces function call routes if the *Maximum route depth* option is greater than 0. If this option was 0 before the profiler started, the Details panel contains no data.

The Call Routes pane contains the following columns:

Columns (in alphabetical order)	**Description**
Hit Count	The number of routine calls per selected route.
Route No	The number of call route.

The Call Stack pane contains the following columns:

Columns (in alphabetical order)	**Description**
Call No	The precedence number. The routine chosen in the Report panel has number 0, its parent has number 1 and so on.
Module Name	Name of the module that holds the routine.
Routine Name	Name of the routine.
Source File	Name of the source file for the routine. The values for this column are read from the application's debug info.
Source Line	Source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.

Call Trace Category

When the Call Trace category is selected in the **Explorer** panel, the Report panel holds results for each routine call. If you select a Report panel row that corresponds to a single routine call, the Function Trace profiler displays information about this call in the Details, Call Tree and Call Graph panels. The Details panel holds two panes: **Routine Parameters On Enter** and **Routine Parameters On Exit**, that show information about parameters used for the routine call. The first one displays parameters that were passed to a routine; the second one displays parameters that were returned by a routine. By comparing the rows of those two panes you can find out what parameters were modified during the routine execution and what are the values of the returned parameters.

Both panes have the same columns (except for the Comment column):

Columns (in alphabetical order)	**Description**
Comment	Extra information about the given parameter. For instance, this column contains a *Return Value* string if the parameter is a result that is returned by the function. This column is displayed for the **Routine Parameters On Exit** pane only.
Param No	The number of the parameter in the parameter list.
Param Type	The parameter's data type.
Param Value	The exact parameter value passed to the selected routine. See *Function Trace Profiler - Tracing Function Call Parameters and Result Values*.

Function Trace Profiler – Outputting Results Using a Custom DLL

The **Function Trace** profiler can display the hierarchy of function calls in the Report panel or in CodeSite. It can also output results to a text file.

If you need special processing for results, you can create a dynamic link library. AQtime loads this DLL upon profiler start and unloads after profiling has finished. The Function Trace profiler sends information about each function call to the library, which, in turn, performs the further result processing.

This dynamic link library must export the following functions:

```
[C++]
void __stdcall Tracer_OnStartProfiling(BOOL Enabled);

void __stdcall Tracer_OnStopProfiling();

void __stdcall Tracer_OnEnabledChanged(BOOL Enabled);

void __stdcall Tracer_OnThreadCreate(LPCWSTR Thread);

void __stdcall Tracer_OnRoutineEnter(LPCWSTR ModuleName,
                                     LPCWSTR ClassName,
                                     LPCWSTR RoutineName,
                                     LPCWSTR Thread,
                                     PAQPROF_PARAMETER_DATA Param);

void __stdcall Tracer_OnRoutineExit (LPCWSTR ModuleName,
                                     LPCWSTR ClassName,
                                     LPCWSTR RoutineName,
                                     LPCWSTR Thread,
                                     PAQPROF_PARAMETER_DATA Param);

void __stdcall Tracer_OnExceptionThrown (LPCWSTR ModuleName,
                                         LPCWSTR ClassName,
                                         LPCWSTR RoutineName,
                                         LPCWSTR Thread);

typedef struct AQPROF_PARAMETER_DATA
  {
    BSTR Type;
    BSTR Value;
    CVOID_PTR pNextParameter;
  } AQPROF_PARAMETER_DATA, *PAQPROF_PARAMETER_DATA;
```

The `Tracer_OnStartProfiling` function is called upon starting the Function Trace profiler. The *Enabled* parameter indicates whether profiling is turned on. Remember that you can switch the profiling state with the **Enable/Disable Profiling** toolbar and menu item or with the `EnableProfiling` function (see *Controlling Profiling From Application Code*). If this occurs the `Tracer_OnEnabledChanged` function is called with *Enabled* to specify the new profiling state. When profiling is finished the `OnStopProfiling` function is called.

The `Tracer_OnThreadCreate` function is called when a new thread is created, the *Thread* parameter contains the thread ID or the thread name (see *Assingning Names to Threads*).

The `Tracer_OnRoutineEnter` function is called when the profiled application enters a function. The `Tracer_OnRoutineExit` function is called when the application exits a function. Both functions have the same parameters that allow AQtime to identify the profiled routine:

Parameter	Description
ModuleName	The name of a module where the profiled routine is declared.
ClassName	The name of a class where the profiled routine is declared.
RoutineName	The name of the profiled routine.
Thread	The ID or name of a thread that called the profiled routine.
Param	A pointer to the `AQPROF_PARAMETER_DATA` structure that holds a linked list of parameters passed to the current routine call. For information about how the parameters are interpreted, see *Function Trace Profiler - Displaying Function Call Parameters* topic. The structure's fields `Type` and `Value` contain the parameter's data type and value, the `pNextParameter` field is a pointer to the next parameter passed to this routine. In case the parameter's value cannot be displayed, the corresponding fields will hold an empty string.

The `Tracer_OnExceptionThrown` function is called upon exception. The values of the *ModuleName, ClassName, RoutineName, Thread* parameters specify the module, class, routine and thread where the exception occured.

For more information on how to create dynamic link libraries, see your development tool's documentation.

Function Trace Profiler Options

The **Function Trace** profiler includes two groups of customizable options:

- One group includes options that have effect on the profiler functioning. Changes in these options will only apply to the next profiler run.
- Another group contains options that affect the currently displayed results. When you change these options, AQtime refreshes the data in its panels.

To modify options that have effect on the way the profiler functions, do the following:

- *AQtime standalone:*
  - Select **Options | Options** from AQtime's main menu (this will call the Options dialog), and then choose **Profilers | Tracing | Function Tracer** from the tree view on the left of the dialog;
  - Press **Configure Current Profiler** on the **Standard** toolbar when the **Function Trace** profiler is selected.
- *AQtime integrated into Visual Studio:*
  - Select **Tools | Options** from Visual Studio's main menu (this will call the Options dialog), and then choose **AQtime | Profilers | Tracing | Function Tracer** from the tree view on the left of the dialog.
- *AQtime integrated into Borland Developer Studio:*
  - Select **Profile | Options** from Borland Developer Studio's main menu (this will call the Options dialog), and then choose **Profilers | Tracing | Function Tracer** from the tree view on the left of the dialog.

Options include:

- **Output**
  - **Get trace info** - Specifies whether additional trace information is collected. The information type is determined by the *Active counter* option. This makes it possible to display trace results in the Details, Call Tree and Call Graph panels.
  - **External DLL output** - If this option is selected, the profiler outputs results using a dynamic link library specified by the *External DLL name* option. For more information, see *Function Trace Profiler - Outputting Results Using a Custom DLL*.
  - **External DLL name** - Specifies the full-path name of the dynamic link library, which AQtime will use to output results of the Function Trace profiler.
  - **Text file output** - Determines whether the Function Trace profiler will output data to text files.
  - **Text output directory** - The path to the folder where the text files will be created.
  - **Text separator** - When writing data to the text file, each function name is preceded by a separator symbol to indicate the depth of nesting. This option specifies this symbol. The following values are available: *None*, *Space*, *Double space* and *Tabulation*.
  - **CodeSite 1 output** - Determines whether CodeSite version 1 is used to display the sequence of function calls in real time.
  - **CodeSite 2/3 output** - Determines whether CodeSite version 2 or 3 is used to display the sequence of function calls in real time.
- **Maximum number of fields** - Specifies how many class fields should be displayed when the class instance is passed as a parameter. Default value is 10. If some fields remain that were not displayed, the value list ends with ellipsis. This option is ignored if the *Maximum route depth* option is set to 0 or if the area's *Retrieve parameter values* property is disabled.
- **Maximum route depth** - Limits the number of function calls to be traced for each route. Default value is 0 and means that the profiler will not trace the routes. Specifying too large value may significantly slow down the profiling.
- **Show multiple functions** - If this option is enabled, there may be multiple function calls in the summary window. This option is ignored if the *Maximum route depth* option is set to 0.
- **Active counter** - Specifies what routine characteristic the profiler will measure. For more information on available values for this option, see *Counters - Overview*.
- **Thread model** - Specifies how the profiler gathers statistics for threads in the profiled application. For more information on available values for this option, see *Profiling Multiple Threads* and *Profiling COM Logical Threads*.

To modify options that affect the result display, use items of the **Profiler** toolbar (by default, it is displayed at the top of the Report panel). If this toolbar is hidden, right-click somewhere in the toolbar area and select Profiler from the subsequent popup list. On the toolbar, the following items are available:

- **Counter unit** - This item is enabled only if the *Active Counter* option is either *Elapsed Time*, *User Time* or *User+Kernel Time*. The *Counter unit* item lets you specify the measurement unit for the time columns in AQtime panels. Possible values are *Seconds*, *Milliseconds*, *Microseconds* and *Machine Cycles*. Note that this option is counter-specific: suppose you browse results of the *User Time* counter and set the option to *Machine Cycles*. If you open the *Elapsed Time* results, change the option to *Seconds* and then return back to the *User Time* results, AQtime will automatically change the option to *Machine Cycles* (that is, it will select the value that was active when you browsed the *User Time* results last time).

- **Show routines with class names** - If this option is enabled, the **Routine Name** column of the Report, Details and Call Tree panels for the Function Trace profiler holds both class name and routine name. Else, this column holds the routine name only.
- **File names with path** - If this option is enabled, the **Source File** and **Module Name** columns of the Report, Details and Call Tree panels for the Function Trace profiler hold the entire path to the given file. Else, these columns hold the file name only.

Light Coverage Profiler

Light Coverage Profiler – Overview

The **Light Coverage** profiler lets you determine whether a routine or a line was executed during the profiler run. This profiler is similar to the **Coverage** profiler, but it *does not* track the hit count, allocate results by threads and trace partially executed lines. This topic provides the Light Coverage profiler overview and describes the profiling results. The complete profiler description includes the following sections:

- *Overview (this topic)*
- *Light Coverage vs. Coverage (this topic)*
- *Description of Profiler Results (this topic)*
- *Description of the Report Panel Columns*
- *Description of the Details Panel Columns*
- *Light Coverage Profiler Options*

Overview

The **Light Coverage** profiler tracks one thing: whether a routine or a line was called during the run. This lets you keep track of untested code as testing progresses over time. It may also let you find unnecessary code that you can remove, once you see that the method or line remains unexecuted under all possible conditions.

The Light Coverage profiler analyzes the application code (32-bit and 64-bit) at two levels of detail: **routine** and **line**. To profile the lines of a routine, you should add this routine to a line-level area (see *Profiling Levels*). Note that to profile managed routines at line level, you have to compile the application with debug information. See *How AQtime Profilers Use Metadata and Debug Information*. If you need to track all lines covered or not covered, begin by using the Light Coverage profiler on the *Full Check* area. This will let you focus on the problem files first, and then you can narrow the analysis to these files and use the Light Coverage profiler with them to drill down further.

The Light Coverage profiler also supports triggers and actions. They allow you to turn the profiling on or off exactly when it is needed during the application run. For more information, see *Using Triggers* and *Using Actions*.

After you have run Light Coverage several times, you can merge profiling results to get mass statistics. Merging can be executed directly from the context menu of the Explorer panel (using the **Merge** item) or done automatically in the background after each profiling run (using the *Auto-merge* option of the Explorer panel). You can also compare results of several Light Coverage runs in order to see the changes. For more information on comparing and merging results, see *Comparing and Merging Results*.

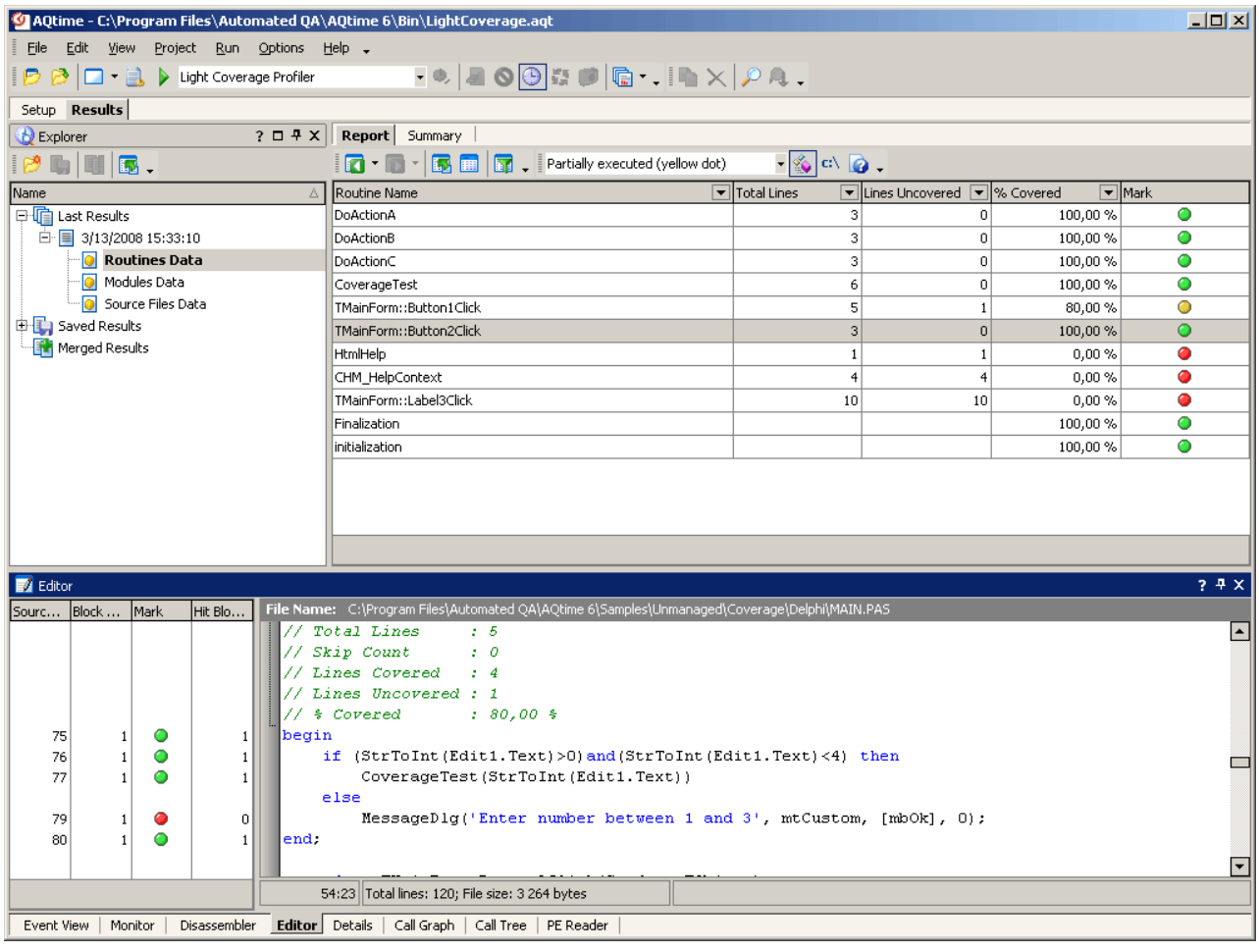
Light Coverage vs. Coverage

The Light Coverage and Coverage profilers perform the same task. When profiling applications, Light Coverage works faster and contributes less resources than the Coverage profiler (This does not work for script profiling though).

Unlike the Coverage profiler, Light Coverage does not collect profiling results by threads. This feature, however, may be a benefit if you use Light Coverage to profile your application when testing it with TestComplete. If you merge the results generated by the Coverage profiler, the number of threads will be increased every time you perform the merge, because each thread has a different name during each application execution and AQtime cannot determine that this thread is the same. So, you have to rename the threads in profiler results before merging them. This problem does not exist with the Light Coverage profiler. It simply does not have threads in results, so merging is easier.

Light Coverage Profiler - Description of Profiler Results

Brief results of the Light Coverage profiler are displayed in the Summary panel. It shows 10 routines that were called most often than other routines and 10 routines that were covered less than other application routines. Information for individual application routines, source files and modules are displayed in the Report panel. Here is an example of a Light Coverage profiler output:

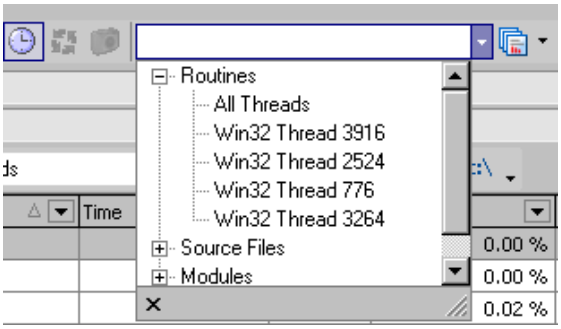

AQtime - C:\Program Files\Automated QA\AQtime 6\Bin\LightCoverage.aqt
File Edit View Project Run Options Help
Light Coverage Profiler
Setup Results
Explorer
Report Summary
Partially executed (yellow dot)
Name
Last Results
3/13/2008 15:33:10
Routines Data
Modules Data
Source Files Data
Saved Results
Merged Results
Routine Name | Total Lines | Lines Uncovered | % Covered | Mark
DoActionA 3 0 100,00 %
DoActionB 3 0 100,00 %
DoActionC 3 0 100,00 %
CoverageTest 6 0 100,00 %
TMainForm::Button1Click 5 1 80,00 %
TMainForm::Button2Click 3 0 100,00 %
HtmlHelp 1 1 0,00 %
CHM_HelpContext 4 4 0,00 %
TMainForm::Label3Click 10 10 0,00 %
Finalization 100,00 %
initialization 100,00 %
Editor
Sourc... Block ... Mark Hit Blo...
File Name: C:\Program Files\Automated QA\AQtime 6\Samples\Unmanaged\Coverage\Delphi\MAIN.PAS
// Total Lines : 5
// Skip Count : 0
// Lines Covered : 4
// Lines Uncovered : 1
// % Covered : 80,00 %
begin
if (StrToInt(Edit1.Text)>0)and(StrToInt(Edit1.Text)<4) then
CoverageTest(StrToInt(Edit1.Text))
else
MessageDlg('Enter number between 1 and 3', mtCustom, [mbOk], 0);
end;
54:23 Total lines: 120; File size: 3 264 bytes
Event View Monitor Disassembler Editor Details Call Graph Call Tree PE Reader

As you can see, the results are organized into three categories:

- Routines
- Source Files
- Modules

The **Source Files** and **Modules** categories let you view summary profiling results for each source file and module in your application. The **Routines** category contains results for each single routine that were included in profiling tasks.

To view profiling results, choose the desired category on the Explorer panel and switch to the Report panel. You can also select the category from the Result Items box of the **Standard** toolbar:



After choosing the desired category, AQtime will update the Report panel. The contents of this panel depend on the currently selected category:

- If you select the **Routines** category, the Report panel will display profiling results one routine per line. Line coverage results will be shown in the **Lines** page of the Details panel and in the Editor's grid.

  We would like to note that to profile code at line level, you should add the desired routines, classes, files or modules to a line-level profiling area (see *Profiling Levels*). Also, to profile managed routines at line level, you have to compile the application with debug information (see *How AQtime Profilers Use Metadata and Debug Information*).

- If you select the **Source Files Data** category, each row in the Report panel will show profiling results for a source file. The Editor panel will display the source code of the selected file. The Details panel will display line profiling results.

  The panel displays results for those lines that were included in profiling tasks (that is, for lines that belong to the routines, classes or files that were included in a profiling area of the line-level type).

- If you select the **Modules** category, each row in the Report panel will display profiling results for one module. The Details and Editor panels will not be used.

The following sections of this topic provides more detailed information about the panels.

Light Coverage Profiler Results - Report Panel

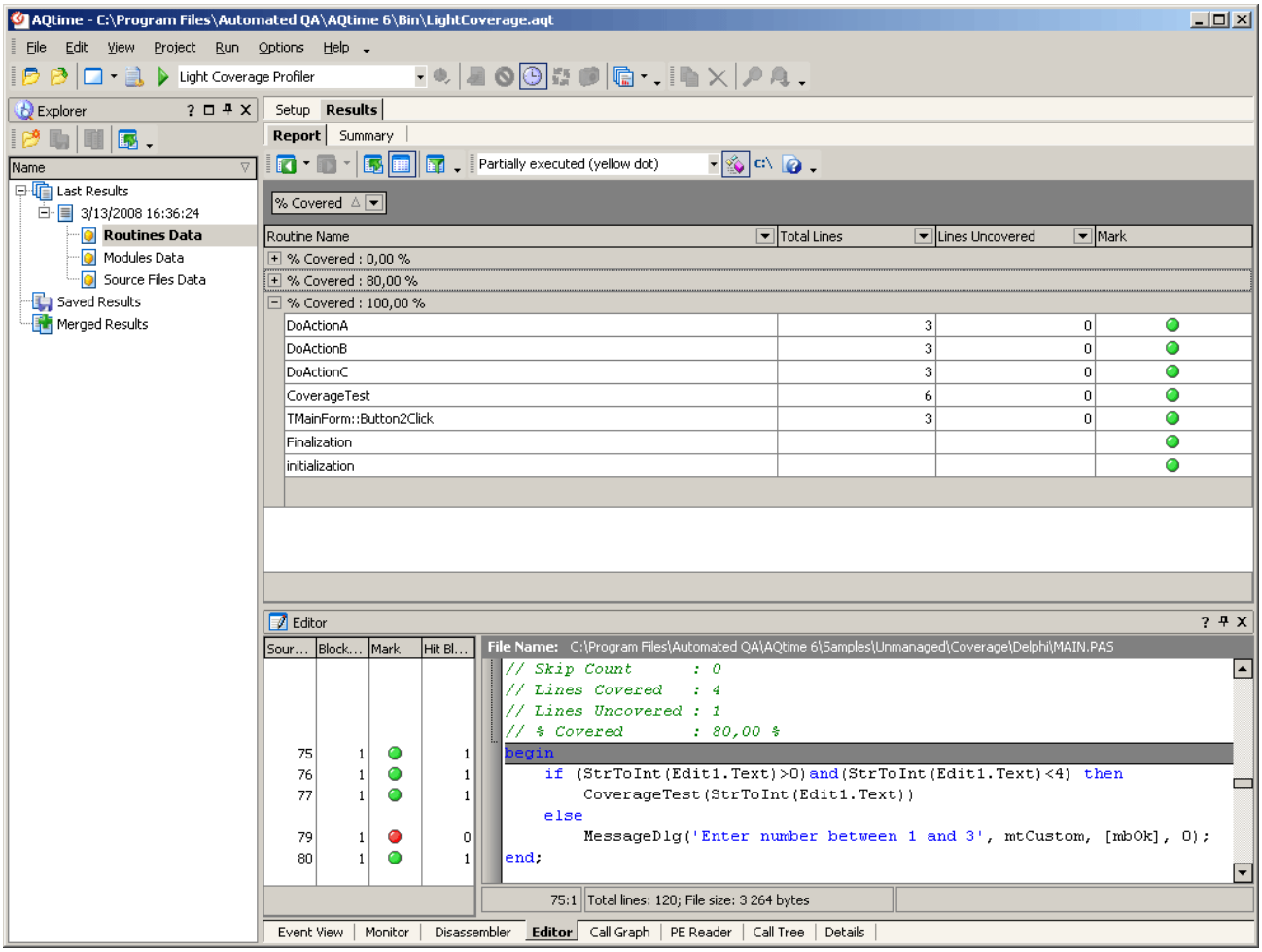
The Report panel displays profiling results according to the category selected in the Explorer panel or in the **Result Items** box on the Standard toolbar. The **Lines Covered**, **Lines Uncovered**, **Total Lines** and **% Covered** columns are used to identify untested code. For instance, if the function includes a large number of lines, most of which only a small percentage were executed during a seemingly "complete" test for the function, you might want to examine the function's algorithm. Of course, AQtime gathers line coverage statistics for those routines that were added to line-level profiling areas. If a routine was profiled at routine level, *% Covered* is either 100%, or 0%. If the Total Lines and the Lines Uncovered values are identical for a routine the routine was not called at all.

You can quickly filter out the routines whose source code lines were covered partially (less than a particular percentage). To do this, use the predefined result views *Routines covered less than %nn*. On the other hand, using the *Unexecuted routines only* predefined view, you can display the routines that were not executed at all. You can select any of these views from the **Result Views** dropdown list on the **Standard** toolbar or from the **View | Result View** menu. If you use AQtime integrated into Microsoft Visual Studio or

Borland Developer Studio, then you can choose the desired view from the **Result Views** dialog. To display it, select **Profile | Result Views** from Visual Studio's menu or click **Result Views** on Borland Developer Studio's **View.AQtime** toolbar. See *Result Views*.

The **Mark** column graphically represents the Light Coverage result. It holds green dots for those routines that were executed during the profiler run and red dots for those routines that were not executed. If a routine was partially executed, the Mark column shows a yellow dot. For complete information on columns, see *Light Coverage Profiler - Report Panel Columns*.

The summary values of the *Lines Covered* and *% Covered* columns display the total number of covered lines and the coverage percentage. To find the percent of covered lines in a class, unit or source file, group results in the Report panel by the **Class Name**, **Unit Name** or **Source File** columns. The group summary will display the coverage results for each class, unit or source file:



The column footer shows summary results for the values displayed in that column. You can customize the summary type and summary format using the Format Columns Dialog. For instance, you can select one of the five summary types (Sum, Count, Avg, Min, Max) or you can hide the summary for the column.

Note that by default the Report panel only shows some of the available columns. You can add more columns to the panel as your needs dictate. For more information on this, see *Adding and Removing Columns*. You can arrange the columns in the panel as needed: move columns, change column width, and so on. For more information on this, see *Arranging Columns, Lines and Panels*.

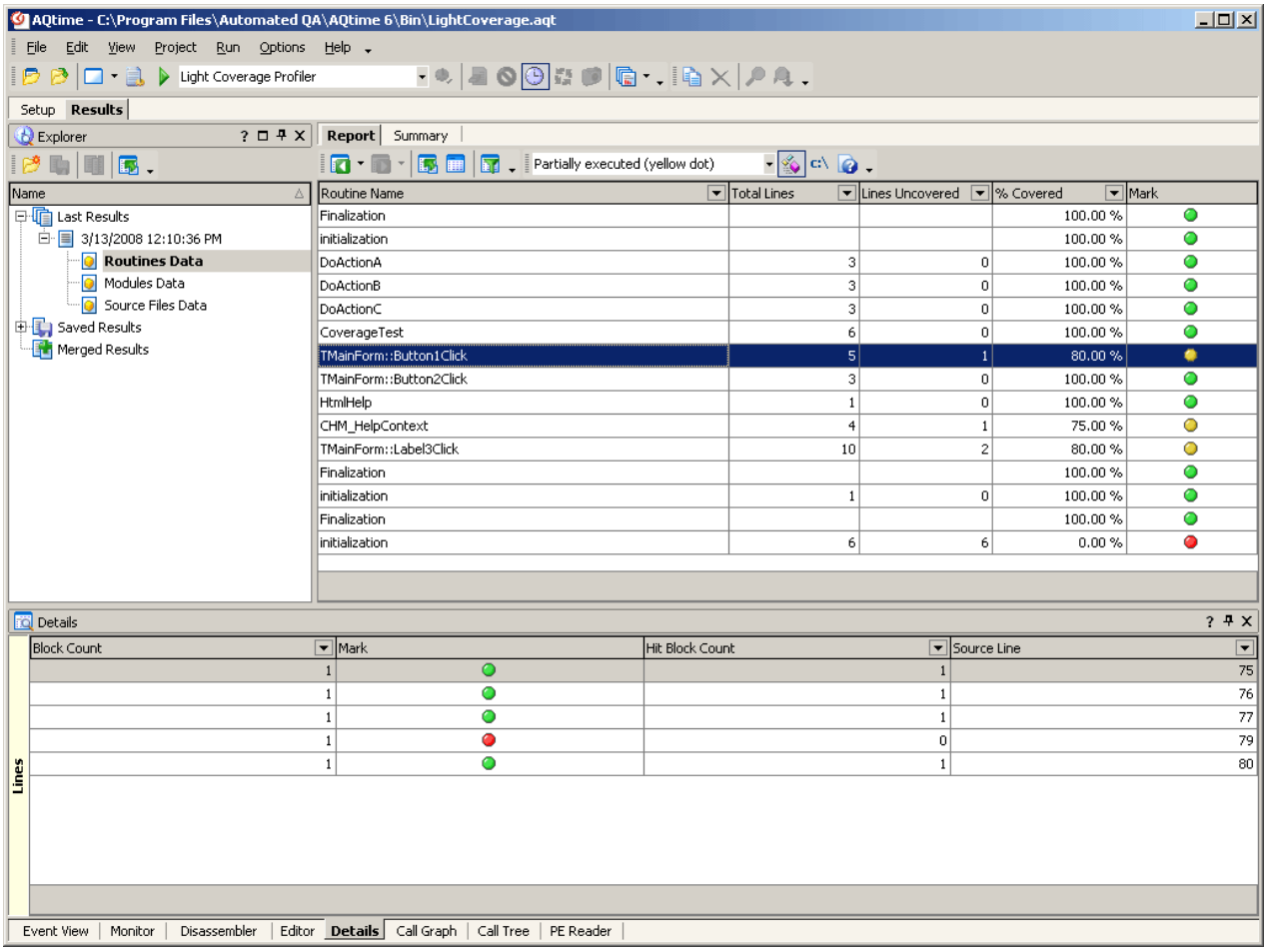
The **Profiler** toolbar contains items that allow you to modify the results that are currently being displayed as your needs dictate. For example, the **Show non-hit routines** toolbar item lets you easily include or

exclude non-executed routines from the result display. For more information on the toolbar items, see *Light Coverage Profiler Options*.

Light Coverage Profiler Results - Details and Editor Panels

The Details and Editor panels display profiling results if you select the *Routines Data* and *Source Files Data* categories. When one of these categories are active, each row in the Report panel corresponds to a single routine or source file of your application. Double-clicking on a routine or file in the Report panel will update the contents of the Details and Editor panels so they will display information concerning that routine or file.

The Details panel holds line coverage profiling results. Each row in this page corresponds to a source code line. You can work with rows on the Lines page the same way you work with rows in the Report panel. For more information on the Details panel columns, see *Light Coverage Profiler - Details Panel Columns*:


AQtime - C:\Program Files\Automated QA\AQtime 6\Bin\LightCoverage.aqt
File Edit View Project Run Options Help
Light Coverage Profiler
Setup Results
Explorer
Report Summary
Partially executed (yellow dot)
Name
Last Results
3/13/2008 12:10:36 PM
Routines Data
Modules Data
Source Files Data
Saved Results
Merged Results
Routine Name | Total Lines | Lines Uncovered | % Covered | Mark
Finalization 100.00 %
initialization 100.00 %
DoActionA 3 0 100.00 %
DoActionB 3 0 100.00 %
DoActionC 3 0 100.00 %
CoverageTest 6 0 100.00 %
TMainForm::Button1Click 5 1 80.00 %
TMainForm::Button2Click 3 0 100.00 %
HtmlHelp 1 0 100.00 %
CHM_HelpContext 4 1 75.00 %
TMainForm::Label3Click 10 2 80.00 %
Finalization 100.00 %
initialization 1 0 100.00 %
Finalization 100.00 %
initialization 6 6 0.00 %
Details
Block Count | Mark | Hit Block Count | Source Line
1 1 75
1 1 76
1 1 77
1 0 79
1 1 80
Lines
Event View Monitor Disassembler Editor Details Call Graph Call Tree PE Reader

The Details panel only displays those lines that belong to routines, classes or source files that were included into line-level profiling areas. For instance, if some routines were included in a line-level area and some were not, the panel will only display those lines that belong to the routines that were included. If your code was profiled at routine level, the panel will be empty.

Note: Compilers can divide some source lines into several blocks. This typically occurs with the `if…then` statements. Suppose, you have the following code:

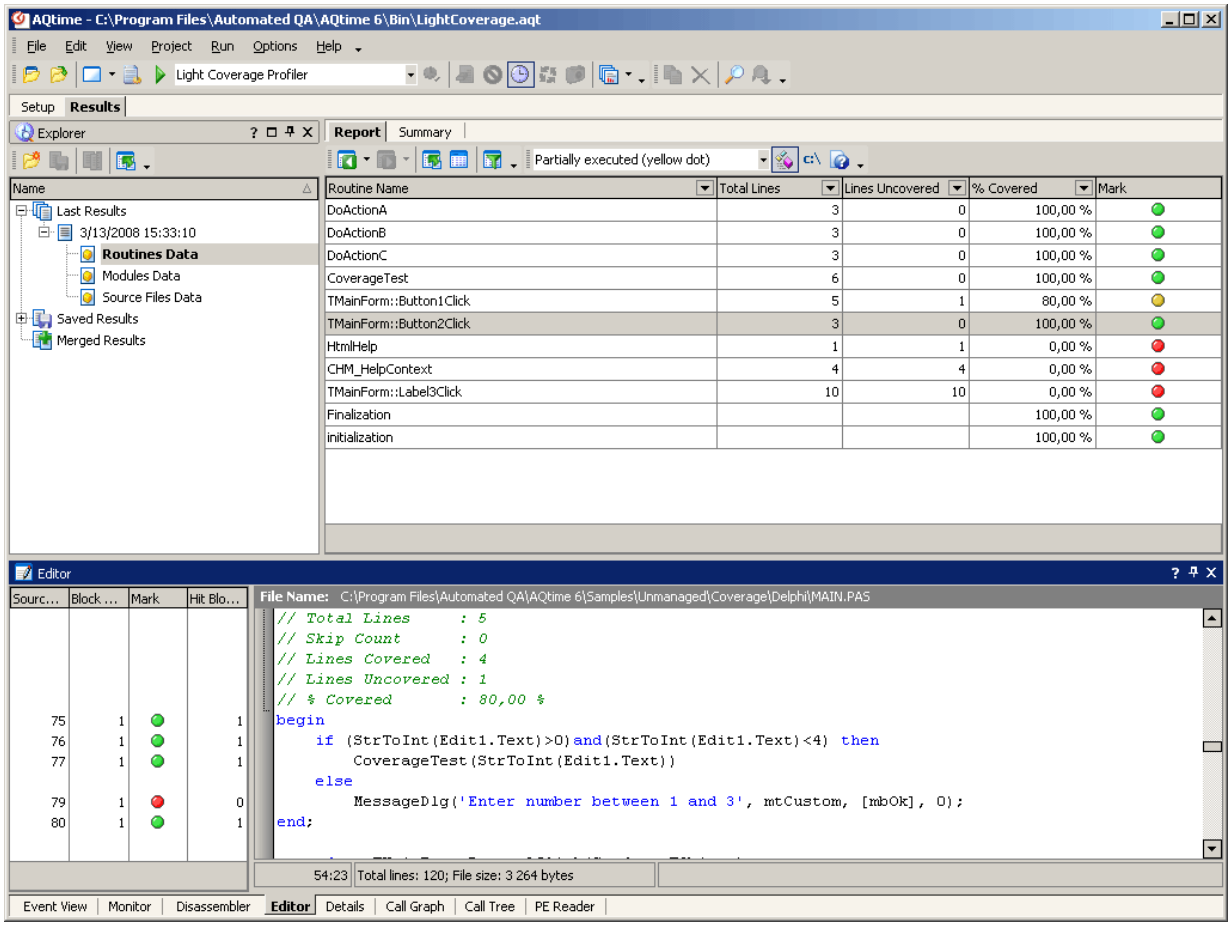
```
if a < b or c > d then DoSomething;
```

This line can be divided into three blocks: 1) the `a < b` condition, 2) the `c > d` condition and 3) call to `DoSomething`. Such lines are called ***partially executed lines***.

The Light Coverage profiler *does not* trace the execution of these blocks. If at least one block was executed, the profiler treats the entire line as executed. To trace partially executed lines, use the Coverage profiler.

Double-clicking a row in the Details grid will move the cursor in the Editor panel to the source code line for which that row displays results. If a routine was profiled at line level, the Editor's grid will show the same results as the ones shown in the Details panel. For instance, you will find that the executed function and lines are marked with green dots and the unexecuted routines and line have red dots.

To select which columns to display in the gutter, use the Field Chooser window. To bring it up, select **Field Chooser** from the context menu. See *Adding and Removing Columns*.


AQtime - C:\Program Files\Automated QA\AQtime 6\Bin\LightCoverage.aqt
File Edit View Project Run Options Help
Light Coverage Profiler
Setup Results
Explorer
Report Summary
Partially executed (yellow dot)
Name
Last Results
3/13/2008 15:33:10
Routines Data
Modules Data
Source Files Data
Saved Results
Merged Results
Routine Name | Total Lines | Lines Uncovered | % Covered | Mark
DoActionA | 3 | 0 | 100,00 %
DoActionB | 3 | 0 | 100,00 %
DoActionC | 3 | 0 | 100,00 %
CoverageTest | 6 | 0 | 100,00 %
TMainForm::Button1Click | 5 | 1 | 80,00 %
TMainForm::Button2Click | 3 | 0 | 100,00 %
HtmlHelp | 1 | 1 | 0,00 %
CHM_HelpContext | 4 | 4 | 0,00 %
TMainForm::Label3Click | 10 | 10 | 0,00 %
Finalization | | | 100,00 %
initialization | | | 100,00 %
Editor
Sourc... Block ... Mark Hit Blo...
File Name: C:\Program Files\Automated QA\AQtime 6\Samples\Unmanaged\Coverage\Delphi\MAIN.PAS
// Total Lines : 5
// Skip Count : 0
// Lines Covered : 4
// Lines Uncovered : 1
// % Covered : 80,00 %
begin
if (StrToInt(Edit1.Text)>0)and(StrToInt(Edit1.Text)<4) then
CoverageTest(StrToInt(Edit1.Text))
else
MessageDlg('Enter number between 1 and 3', mtCustom, [mbOk], 0);
end;
54:23 Total lines: 120; File size: 3 264 bytes
Event View Monitor Disassembler Editor Details Call Graph Call Tree PE Reader

Note: The Code Editor of Microsoft Visual Studio or Borland Developer Studio lets you collapse and expand blocks of source code. The grid, which AQtime adds to the Code Editor to display profiling results, supports neither collapsing, nor expanding, because Visual Studio (or Developer Studio) does not send appropriate notifications to AQtime. So, to ensure that the grid shows proper profiling results for source lines and routines, please expand all of the collapsed blocks of code. To do this in Microsoft Visual Studio, use the **Outlining | Toggle All Outlining** or **Outlining | Stop Outlining** item of the Code Editor's context menu. To do this in Borland Developer Studio, use the **Unfold | All** item of the Editor's context menu.

Some notes on displaying results in the Editor panel:

- Note that in order for AQtime to show source files in the Editor, the path to these files must be specified in the Project Search Directories or Search Directories dialogs. In addition, your applications must be compiled with debug information (see *How AQtime Profilers Use Metadata and Debug Information*).
- Information about lines depends on debug info attached to the executable. With the Coverage profiler especially, you should be on the lookout for unexpected discrepancies. Some compilers, for instance, such as Borland Delphi, will skip functions that are never called (this is called Smart Linking). Therefore, the debug information will log fewer functions and fewer lines than there are in the source file.
- The Editor can display incorrect profiling-related information for some C++ applications that use several functions based on the same template (see *Profiling Template Functions*). In this case, refer to the Report panel to get the correct results.
- Sometimes the Light Coverage profiler can report that the last line of your routine was not executed (it shows a red dot for this line in the Editor's grid). The reason for this is that AQtime stops profiling a routine when the application executes the `ret` instruction. Usually, this instruction is the last instruction in the routine's binary code. However, compilers can produce code which includes `ret` instructions in "the middle" of a routine. For example, the following C++Builder code will insert `ret` instructions after each `case` line of a `switch...case` block.

```
[C++]
void foo(int i)
{
  ...
  switch(i)
  {

    case 1: //do something
      break;
    case 2: // do something
      break;
  }
} // This line is never executed
```

Light Coverage Profiler - Report Panel Columns

When displaying results of the **Light Coverage** profiler, each row in the **Report** panel holds the results for a routine, source file or module in your application. Which values are displayed depends on the category selected in the **Explorer** panel:

- Routines Category
- Source Files and Modules Category

For more information on the categories, see *Light Coverage Profiler - Description of Results*.

Note that the Light Coverage profiler only shows some of the available columns in the Report panel by default. You can add more columns to the panel as needed. For more information, see *Adding and Removing Columns*.

Routines Category

Columns (in alphabetical order)	**Description**

Analysis Result	Specifies if the routine was instrumented or not. If the routine was instrumented, this column is empty. Otherwise, the column displays a short description of why the routine was not instrumented: • *Less than 5 bytes* - The routine occupies less than 5 bytes in memory. See *Profiling Small Functions*. • *No line info* - The routine was added to a line-level area, but the debug information does not hold info about routine lines. These routines can be profiled at routine level only. • *Unsafe code* - AQtime was not able to instrument the routine safely. This typically occurs when the routine's binary code is intermixed with data areas. See *Profiling Routines That Hold Unsafe Code*. • *No ret instruction* - The routine's binary code does not contain the `ret` instruction (this may happen if the routine finishes with the `jmp` instruction). See *Profiling Routines That Do Not Have the Ret Instruction*. • *Duplicated code* - The routine whose code coincides with the code of another routine. To learn more about this, see *Profiling Duplicated Code*.
Class Name	If the routine is a method, this is the name of the class to which it belongs.
Code Type	Specifies the type of the routine's code. The following values are possible: • **MSIL** - Managed-code routine with MSIL (Microsoft Intermediate Language) code. • **x64** - 64-bit code routine. • **x86** - Native-code (unmanaged) routine. • **Pseudo** - Pseudo routine that was created by the context. For example, *<JIT Compiler>*, *<Garbage Collector>*, *<Unknown PInvoke>* or *<Root>*. • **PInvoke** - Native-code routine for which there is a declaration in one of the managed modules and that is called from within the unmanaged code. • **NGen** - Managed routine that was compiled by the *ngen* utility (CLR Native Image Generator) with the */prof* argument in its command line. The *ngen* compilation means that the routine was compiled before the application starts. • **Script** - The routine belongs to a script that was profiled along with the host application. See *Profiling Scripts* for details.
Lines Covered	Specifies the number of routine's source lines that were executed during the profiler run. **Note:** Partially executed lines are counted as executed.
Lines Uncovered	Specifies the number of routine's source lines that were not executed during the profiler run.

Mark	Specifies if the routine was executed or not during the profiler run. If the routine was profiled at routine level, Mark holds a green dot if the method was executed or a red dot if the method was not executed. If the method was profiled at line level, Mark holds a green dot if all lines of the method were executed, a red dot if no lines of the method was executed and a yellow dot if some lines were executed and some were not.
Module Name	The name of the module which contains the profiled routine.
Routine Name	Name of the routine.
Source File	Name of the source file for the method. The values for this column are read from the application's debug info. If debug info does not contain information on the file name, the column is empty.
Source Line	Source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.
Total Lines	The total number of source code lines in the routine.
% Covered	The percentage of covered lines against the total number of lines in the routine. Partially executed lines are counted as executed.

Source Files and Modules Categories

Columns (in alphabetical order)	**Description**
File Name or **Module Name**	Name of the source file (or module). The source file name is read from the application's debug info. If debug info does not contain information on the file name, the column is empty.
% Covered	The percentage of covered lines against the total number of lines in the routines that belong to the source file (module). Partially executed lines are counted as executed.

Light Coverage Profiler - Details Panel Columns

If a routine was profiled at routine level, the Light Coverage profiler uses the Details panel to display additional profiling results for that routine. The panel holds the **Lines** table that shows the line profiling results for the routine selected in the Report panel. This table is empty if the routine was not profiled at line level. Each row in the table holds profiling results for a source code line:

Columns (in alphabetical order)	**Description**
Mark	Specifies whether the source line was executed or not. If the line was executed, the Mark column holds a green dot. If the line was not executed, Mark holds a red dot. Partially executed lines are counted as executed.
Source Line	The line number in the source file.

Light Coverage Profiler Options

The Light Coverage profiler contains options that affect how the current results display. If you change these options, AQtime will refresh data in its panels.

To modify these options, use the items of the **Profiler** toolbar. If it is hidden, right-click somewhere within the toolbar area and select **Profiler** from the subsequent popup list. The toolbar contains the following items:

- **Mark partially executed lines as** - Specifies how AQtime treats partially executed lines when it is calculating the number of covered and un-covered lines and how AQtime marks them (with a red, green or a yellow dot) in the **Lines** pane of the Details panel and in the Editor. This option can have one of three values:

Value	Description
Partially executed	AQtime marks partially executed lines with yellow dots and treats them as non-executed when calculating the **Lines Covered**, **Lines Uncovered** and **% Covered** values.
Completely executed	AQtime marks partially executed lines with green dots and treats them as executed when calculating the **Lines Covered**, **Lines Uncovered** and **% Covered** values.
Non-executed	AQtime marks partially executed lines with red dots and treats them as non-executed.

- **Show routines with class names** - If it is enabled, the **Routine Name** column of the Report panel for the Coverage profiler displays the name of the given routine along with the name of the class the routine belongs to. Otherwise, this column only displays the routine name.
- **File names with path** - If this option is enabled, the **Source File** and **Module Name** columns of the Report panel for the Light Coverage profiler hold the entire path to the given file. Else, these columns hold the file name only.

Load Library Tracer

Load Library Tracer – Overview

The **Load Library Tracer** determines what dynamic link libraries are being loaded and unloaded by the profiled application and how many times they were loaded and unloaded. Loading and unloading of a dll multiple times can significantly slow down the application, and this profiler can help you find such problems. This topic provides the Load Library Tracer overview. The complete profiler description includes the following topics:

- *Overview (this topic)*
- *Description of the Report Panel Columns*
- *Description of the Details Panel Columns*
- *Load Library Tracer Options*

Overview

As we have said, loading and unloading of a DLL multiple times during the application execution can significantly slow down the application. To understand why this may happen, let's explain how the dynamic link libraries are used.

When an application loads a dynamic link library for the first time, the operating system loads the library into memory and returns the library's handle to the application. If the application attempts to load the library

once again, the operating system does not load the library code in memory for the second time. It just increases the reference counter for the library and returns the the library's handle to the application.

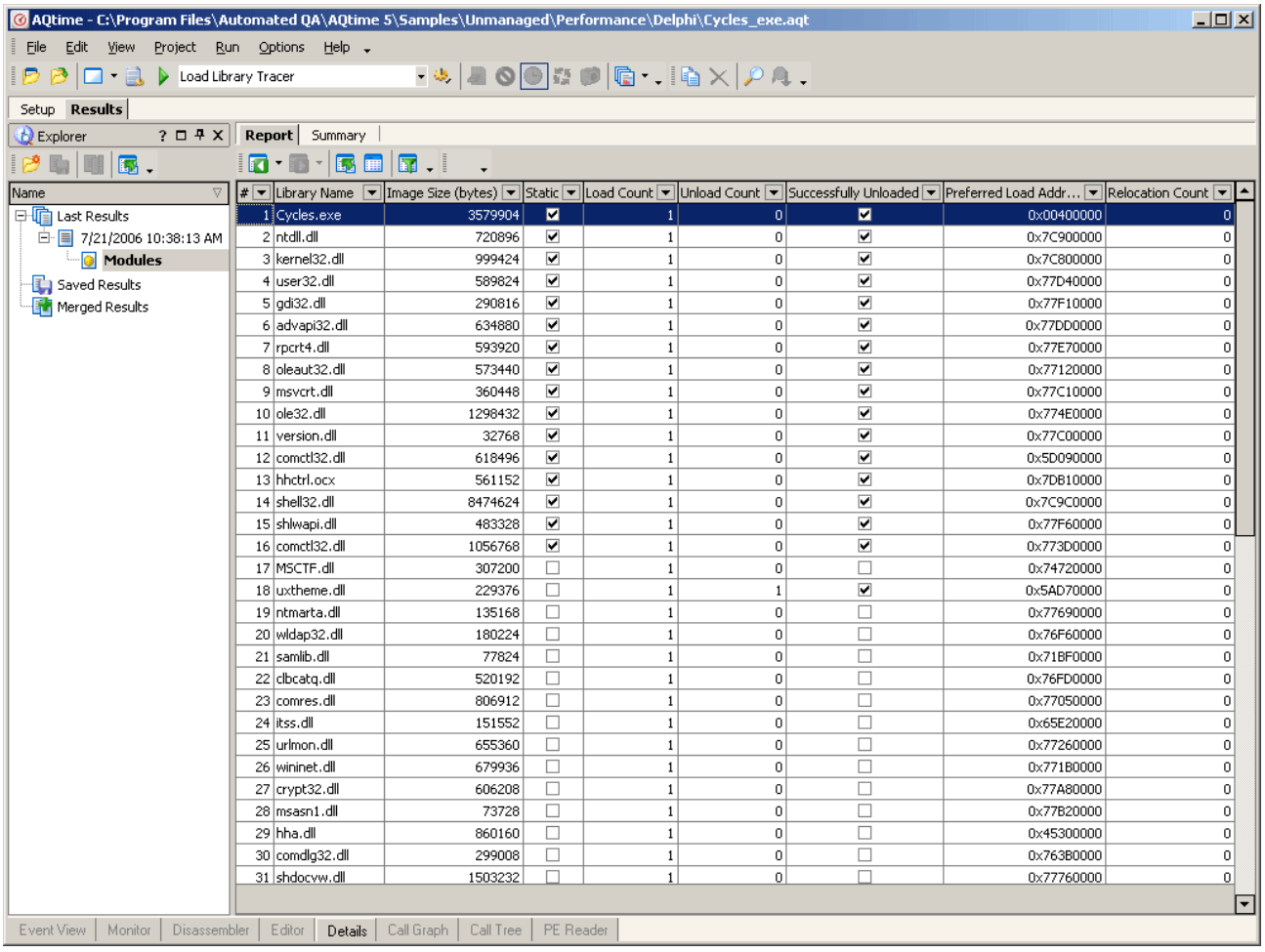
When an application unloads the library, the operating system decreases the reference counter and when the counter value is 0, the operating system unloads the library from memory.

Now imagine a situation that some function within an application loads a DLL in memory, calls DLL functions and then unloads the DLL as it is not needed anymore. Some time later another function written by another developer loads the same DLL in memory, calls some DLL routines and then unloads the DLL. Loading and unloading of dynamic link libraries requires some time and doing these multiple times will decrease the overall application performance.

With the Load Library Tracer you can easily detect such situations and optimize your code. The Load Library Tracer supports both 32- and 64-bit applications. It analyzes DLLs loaded both at load time and run time.

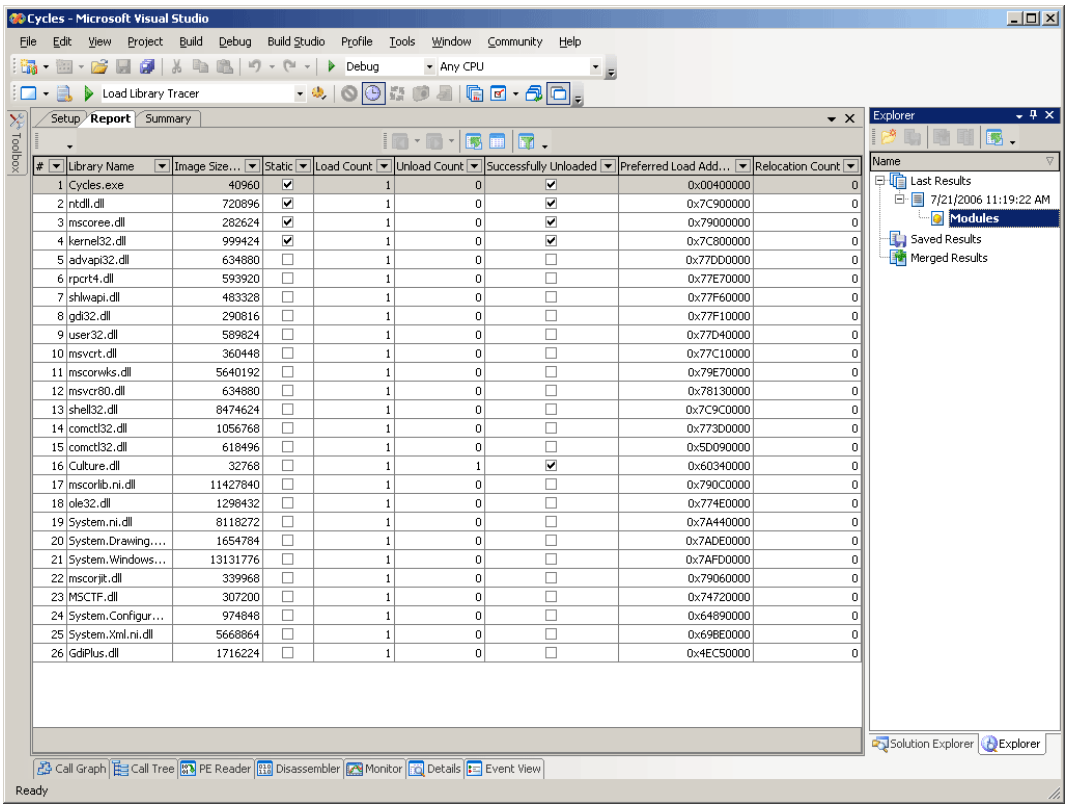
The profiler reports about all modules used by the tested application, even the modules that are not directly loaded by the application itself. For example, if you profile an executable that loads Lib1.dll, which in turn loads Lib2.dll, the profiler will include both Lib1 and Lib2 into the report.

Here is a sample profiler output:

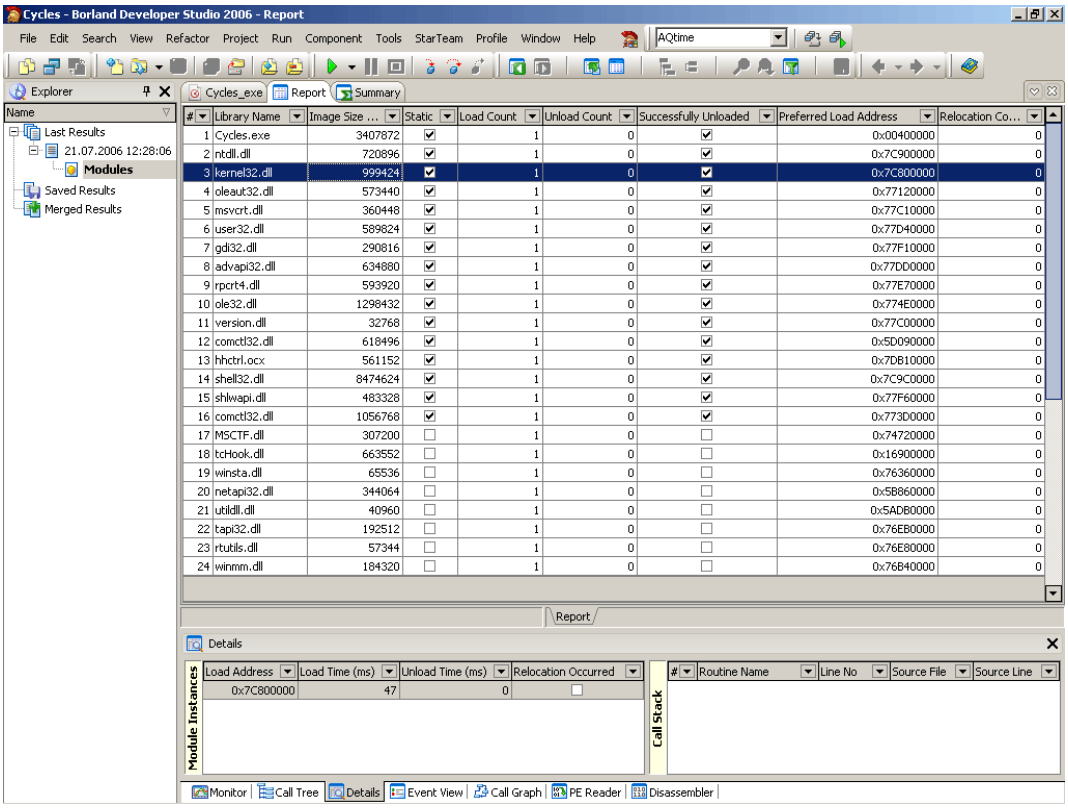


#	Library Name	Image Size (bytes)	Static	Load Count	Unload Count	Successfully Unloaded	Preferred Load Addr...	Relocation Count
1	Cycles.exe	3579904	☑	1	0	☑	0x00400000	0
2	ntdll.dll	720896	☑	1	0	☑	0x7C900000	0
3	kernel32.dll	999424	☑	1	0	☑	0x7C800000	0
4	user32.dll	589824	☑	1	0	☑	0x77D40000	0
5	gdi32.dll	290816	☑	1	0	☑	0x77F10000	0
6	advapi32.dll	634880	☑	1	0	☑	0x77DD0000	0
7	rpcrt4.dll	593920	☑	1	0	☑	0x77E70000	0
8	oleaut32.dll	573440	☑	1	0	☑	0x77120000	0
9	msvcrt.dll	360448	☑	1	0	☑	0x77C10000	0
10	ole32.dll	1298432	☑	1	0	☑	0x774E0000	0
11	version.dll	32768	☑	1	0	☑	0x77C00000	0
12	comctl32.dll	618496	☑	1	0	☑	0x5D090000	0
13	hhctrl.ocx	561152	☑	1	0	☑	0x7DB10000	0
14	shell32.dll	8474624	☑	1	0	☑	0x7C9C0000	0
15	shlwapi.dll	483328	☑	1	0	☑	0x77F60000	0
16	comctl32.dll	1056768	☑	1	0	☑	0x773D0000	0
17	MSCTF.dll	307200	☐	1	0	☐	0x74720000	0
18	uxtheme.dll	229376	☐	1	1	☑	0x5AD70000	0
19	ntmarta.dll	135168	☐	1	0	☐	0x77690000	0
20	wldap32.dll	180224	☐	1	0	☐	0x76F60000	0
21	samlib.dll	77824	☐	1	0	☐	0x71BF0000	0
22	clbcatq.dll	520192	☐	1	0	☐	0x76FD0000	0
23	comres.dll	806912	☐	1	0	☐	0x77050000	0
24	itss.dll	151552	☐	1	0	☐	0x65E20000	0
25	urlmon.dll	655360	☐	1	0	☐	0x77260000	0
26	wininet.dll	679936	☐	1	0	☐	0x771B0000	0
27	crypt32.dll	606208	☐	1	0	☐	0x77A80000	0
28	msasn1.dll	73728	☐	1	0	☐	0x77B20000	0
29	hha.dll	860160	☐	1	0	☐	0x45300000	0
30	comdlg32.dll	299008	☐	1	0	☐	0x763B0000	0
31	shdocvw.dll	1503232	☐	1	0	☐	0x77760000	0

AQtime standalone



AQtime integrated into Visual Studio



AQtime integrated into Borland Developer Studio

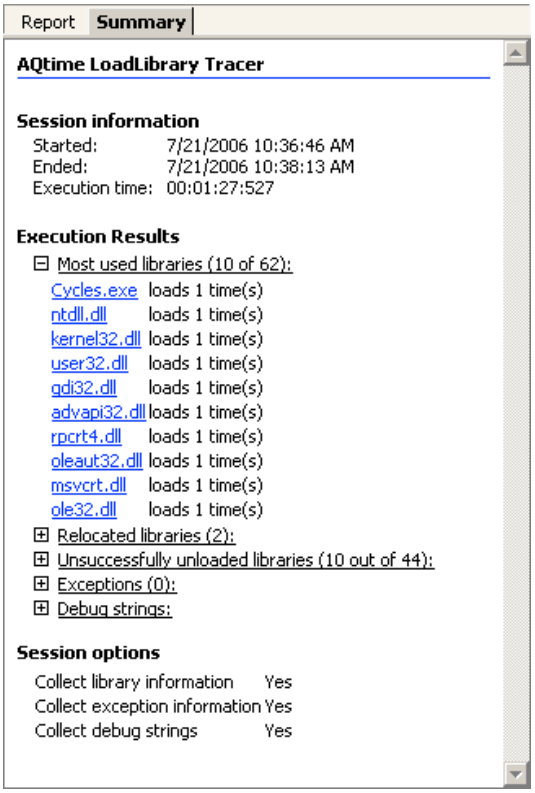
As you can see, the **Report** panel displays modules loaded to and unloaded from memory during the profiling time. The results include the active module of your AQtime project, since this module is also loaded to and unloaded from memory at profiling time.

The panel columns indicate the module name, number of loads and unloads, file size and other information on the libraries. To determine DLLs loaded muptiple times, analyze the Load Count column. This column displays how many times the library was loaded in memory. Large values in this column point to possible ineffective usage of the DLL.

For information on other Report panel columns, see *Load Library Tracer - Report Panel Columns*.

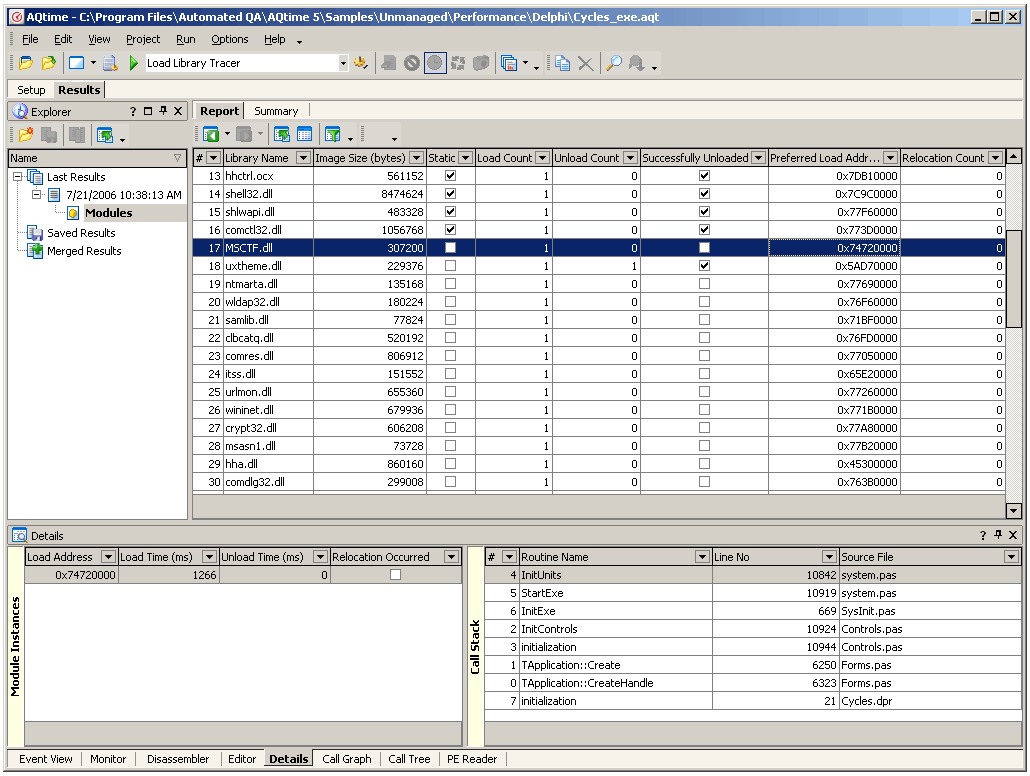
Note that the Report panel does not contain all the available columns by default. You can add columns to the panel at your desire. See also *Arranging Columns, Lines and Panels* for information on how you can tune AQtime panels.

The most used libraries are also displayed on the **Summary** panel:



To view detailed results for a library, just click its name in the Summary panel. AQtime will bring to front the Report panel and select the corresponding row in it.

To view information on library loading, double-click the desired library in the Report panel and swtich to the **Details** panel. This panel contains two tables: Module Instances and Call Stack:


AQtime - C:\Program Files\Automated QA\AQtime 5\Samples\Unmanaged\Performance\Delphi\Cycles_exe.aqt
File Edit View Project Run Options Help
Load Library Tracer
Setup Results
Explorer
Report Summary
Name
Last Results
7/21/2006 10:38:13 AM
Modules
Saved Results
Merged Results
Library Name Image Size (bytes) Static Load Count Unload Count Successfully Unloaded Preferred Load Addr... Relocation Count
13 hhctrl.ocx 561152 1 0 0x7DB10000 0
14 shell32.dll 8474624 1 0 0x7C9C0000 0
15 shlwapi.dll 483328 1 0 0x77F60000 0
16 comctl32.dll 1056768 1 0 0x773D0000 0
17 MSCTF.dll 307200 1 0 0x74720000 0
18 uxtheme.dll 229376 1 1 0x5AD70000 0
19 ntmarta.dll 135168 1 0 0x77690000 0
20 wldap32.dll 180224 1 0 0x76F60000 0
21 samlib.dll 77824 1 0 0x71BF0000 0
22 clbcatq.dll 520192 1 0 0x76FD0000 0
23 comres.dll 806912 1 0 0x77050000 0
24 itss.dll 151552 1 0 0x65E20000 0
25 urlmon.dll 655360 1 0 0x77260000 0
26 wininet.dll 679936 1 0 0x771B0000 0
27 crypt32.dll 606208 1 0 0x77A80000 0
28 msasn1.dll 73728 1 0 0x77B20000 0
29 hha.dll 860160 1 0 0x45300000 0
30 comdlg32.dll 299008 1 0 0x763B0000 0
Details
Module Instances
Load Address Load Time (ms) Unload Time (ms) Relocation Occurred
0x74720000 1266 0
Call Stack
Routine Name Line No Source File
4 InitUnits 10842 system.pas
5 StartExe 10919 system.pas
6 InitExe 669 SysInit.pas
2 InitControls 10924 Controls.pas
3 initialization 10944 Controls.pas
1 TApplication::Create 6250 Forms.pas
0 TApplication::CreateHandle 6323 Forms.pas
7 initialization 21 Cycles.dpr
Event View Monitor Disassembler Editor Details Call Graph Call Tree PE Reader

Each row of the Module Instances table corresponds to the loading of the DLL in memory. When you select a row within this table, the Call Stack displays the sequence of function calls that led to loading the library in memory.

For information on table columns, see *Load Library Tracer - Details Panel Columns*.

Load Library Tracer – Report Panel Columns

When you view results of the Load Library Tracer, the **Report** panel displays the list of modules (DLLs) loaded in memory during the profiling time. The panel also displays the active module of your AQtime project as it is also loaded in memory during the profiling. The panel contains the following columns:

Columns (in alphabetical order)	**Description**
#	The index of the library. Indicates the sequence in which libraries were loaded in memory.
Image Size (bytes)	The size of the library code in memory.
Library Name	The name of the dynamic link library.
Library Path	Path to the library file.
Load Count	Specifies how many times the library was loaded in memory.
Preferred Load Address	The preferred load address specified in the DLL header. The actual address where the dll was loaded is specified in the Load Address column of the Details panel and in the Event View panel of the message that informs that the library was loaded. Be aware that if the actual address where Windows loads the DLL differs from the preferred address, there will be a time penalty for the required relocations by Windows at run time.
Relocation Count	Specifies the number of time the DLL was relocated in memory. The

	relocation, for instance, occurs when the library is not loaded at the preferred address specified in its header.
Size (bytes)	Size of the library file in bytes.
Static	If this column is checked, the library was loaded by the executable at the load time. Else, it was loaded dynamically (at run time).
Successfully Unloaded	Additional indicator that specifies if Load Count is equal to Unload Count. If results were generated upon closing the profiled application, this column is always checked. It can be unchecked, if you get results during the profiling through **Run | Get Results** (**Profile | Get Results** in Visual Studio or **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar).
Unload Count	Specifies how many times the library was unloaded. Usually, this value equals to Load Count when the application is closed. It can differ from Load Count when you get results during the profiling through **Run | Get Results** (**Profile | Get Results** in Visual Studio or **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar).

Load Library Tracer – Details Panel Columns

When you review the Load Library Tracer results, the Report panel displays information on libraries that were loaded in memory at profiling time. The **Details** panel contains two tables - Module Instances and Call Stack - display information on loads of the selected library (see profiler description).

Each row of the **Module** Instances table corresponds to the library load in memory. The panel contains the following columns:

Columns (in alphabetical order)	**Description**
Delta (ms)	Specifies the difference between the Unload Time and Load Time values. In other words, this column specifies the number of milliseconds, which the module instance existed in memory. If the unload time cannot be determined, Delta is 0.
Load Address	Specifies the address, at which the library was loaded.
Load Time (ms)	Specifies the number of milliseconds that have passed since the profiler started and the module was loaded in memory.
Relocation Occurred	Specifies whether the operating system relocated the library in memory during the load. The relocation occurs, for example, when the operating system loads the library to another address rather than the address specified in the library header.
Thread ID	Specifies the identifier of the thread that contains the code that loaded the module in memory.
Unloaded Thread ID	Specifies the identifier of the thread that contains the code that unloaded the module from memory.
Unload Time (ms)	Specifies the number of milliseconds that have passed since the profiler started and the module was unloaded from memory. If the unload time cannot be determined, this column displays 0.

Note: If you use a computer that has several processors or a multiple-core processor (for example, dual-core CPU) and has Windows XP Service Pack 2, then you must install the

Windows update #896256 in order for the profiler to be able to time your application correctly. The update is available on Microsoft's web site:

- http://support.microsoft.com/kb/896256

The **Call Stack** panel displays the sequence of function calls that led to loading the library in memory. Each row in the panel corresponds to a routine. The panel contains the following columns:

Columns (in alphabetical order)	**Description**
#	Index of routine in the call stack. The function with index 0 contains calls to API functions that loaded the library in memory.
Class Name	The name of the class to which the routine belongs.
Line No	Number of the source line holding the call to the next routine (the routine having lower index) in the stack.
Module Name	The name of the module that contains the routine's code.
Namespace	The name of the namespace, to which the routine's source code belong.
Routine Name	The name of the routine.
Source File	Name of the source file that contains the routine's source code.
Source Line	The source line number in the source file, at which the routine's code begins.
Token	The routine's token.
Unit Name	Name of the unit holding the routine's source code.

Note that AQtime retrieves some part of information displayed in the Call Stack from the module's debug information. So, if the module does not have debug information or the debug information does not contain needed data, some of the Call Stack columns can be empty.

Load Library Tracer Options

The Load Library Tracer includes one option only:

- **Stack depth** - Specified the maximum number of traced routines in the stack. Default value is 80.

To change the option, do any of the following:

- *AQtime standalone:*
  - Select **Options | Options** from AQtime's main menu (this will call the Options dialog) and then choose **Profilers | Tracing | Load Library Tracer** from the tree view on the left of the dialog;
  - Press **Configure Current Profiler** on the **Standard** toolbar when the **Load Library Tracer** profiler is selected.
- *AQtime integrated into Visual Studio:*
  - Select **Tools | Options** from Visual Studio's main menu (this will call the Options dialog) and then choose **AQtime | Profilers | Tracing | Load Library Tracer** from the tree view on the left of the dialog.
- *AQtime integrated into Borland Developer Studio:*

- Select **Profile | Options** from Borland Developer Studio's main menu (this will call the Options dialog) and then choose **Profilers | Tracing | Load Library Tracer** from the tree view on the left of the dialog.

Reference Count Profiler

Reference Count Profiler – Overview

The **Reference Count** profiler tracks the number of references to interface objects. This topic provides the Reference Count profiler overview. The complete profiler description includes the following topics:

- *General Information (this topic)*
- *Profiler Description (this topic)*
- *Description of the Report Panel Columns*
- *Description of the Details Panel Columns*
- *Reference Count Profiler Options*

General Information

The COM (Component Object Model) specifications require that individual objects remain alive as long as there are clients which have acquired access to one or more of its interfaces. Moreover, the COM object should be properly disposed when all code that used the object have finished and the object is no longer utilized.

To fulfill these requirements the reference counting technique is applied. The general idea of reference counting is as follows:

- Each COM object maintains a special variable that stores the current number of references to the object.
- When a new client acquires access to object's interface, the reference counter is increased.
- When the client has finished using the COM object, the counter is decreased. If the reference counter equals zero, then the object is no longer required and can be released.

To implement reference counting every COM object supports the IUnknown interface or its descendants. The interface declares the the AddRef and Release methods that increment/decrement the object reference counter.

The purpose of `AddRef` is to indicate to the COM object that an additional reference to the object has been added, and hence it is necessary to remain alive as long as this reference is still valid. Conversely, the purpose of `Release` is to indicate to the COM object that a client (or a part of the client's code) has no further need for it and hence if this reference count has dropped to zero, it may be time to destroy itself.

Some high-level programming languages, like Delphi and Visual Basic, provide automatic reference counting. In these languages, a new reference is added automatically when a new object is created. And when the object goes beyond its visibility scope, the references to it are automatically removed. Thus, COM objects are typically used without explicit calls to the `AddRef` and `Release` methods. Though still there can be situations when you need to call these methods manually. In another programming languages, for example, in C++, the programmer should care of reference counting all by himself.

In any case, when these methods are called manually, each `AddRef` should have a matching Release call. If the number of `AddRef` calls exceeds the number of `Release` calls, the interface object will never be freed, thus producing a memory leak. If `Release` is called more times than `AddRef`, the reference object will be destroyed prematurely. In the latter case, an access violation will occur when a non-existing interface object is addressed.

Therefore, the code snippets in which `AddRef` and `Release` are called explicitly should be inspected first if you encounter problems with interface objects. The Reference Count profiler would be useful in this situation.

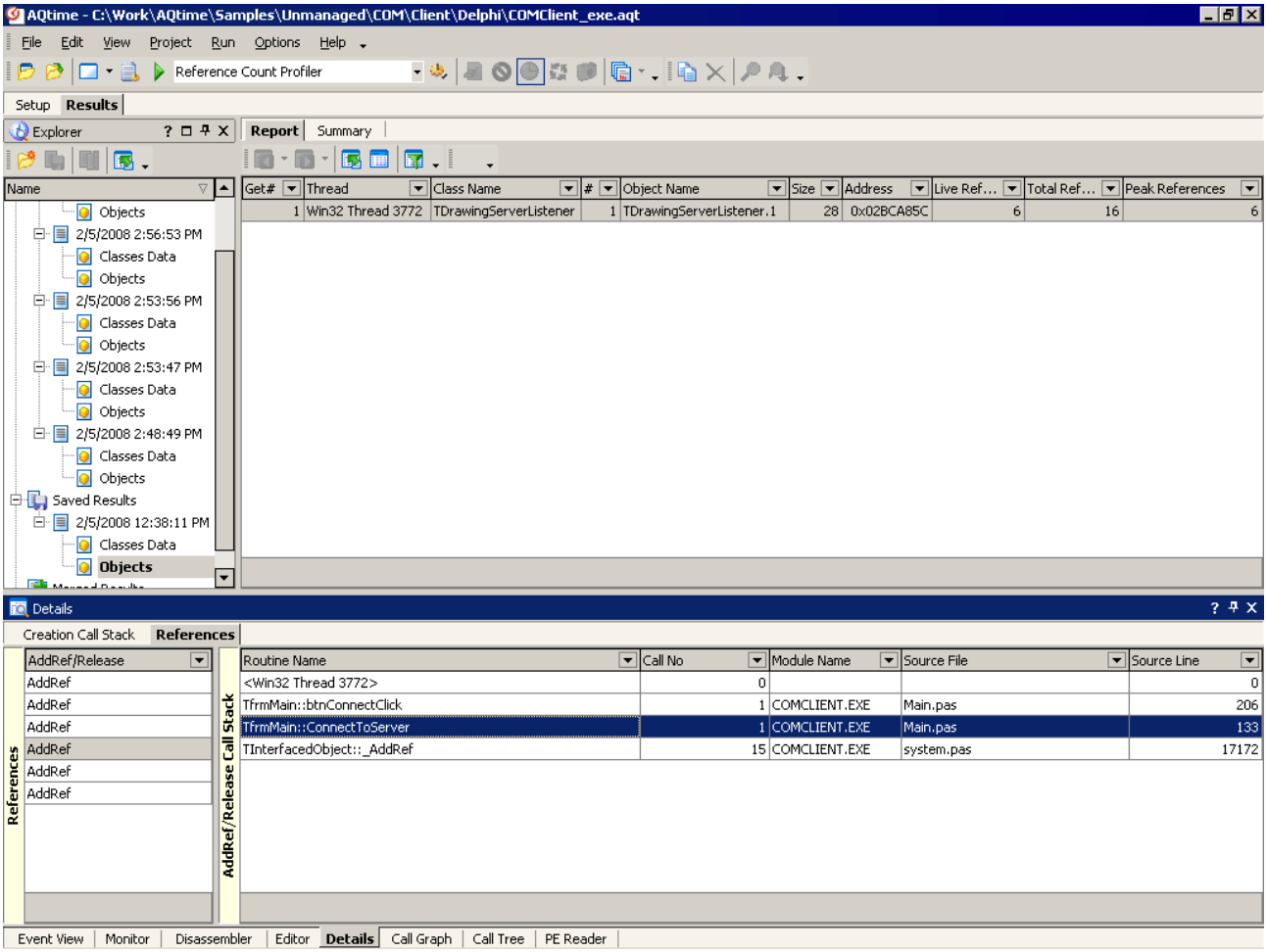
Profiler Description

The Reference Count Profiler analyzes the use of objects that implement the `IUnknown` interface or its descendants. The main goal of the profiler is locate unreleased or prematurely released interface objects. It logs calls to the `AddRef` and `Release` methods, traces their call stack and reports how many references were used for each object, in total and as a peak count for the run.

The Reference Count profiler operates during the entire run of the application. It takes no account of triggers and actions and disables the **Enable/Disable Profiling** button.

The profiling results are displayed in the Report panel and are organized into the **Classes Data** and **Objects** categories. The first category provides a general overview of which classes produce interface objects, as well as information about the total, peak or current number of object references to this class. It reports data about every interface class that was utilized during the whole runtime of the application, no matter whether it existed when the results were retrieved. The Objects category provides a more detailed report on each interface object that existed at time when results were generated. See *Report column descriptions* for a full list of displayed columns.

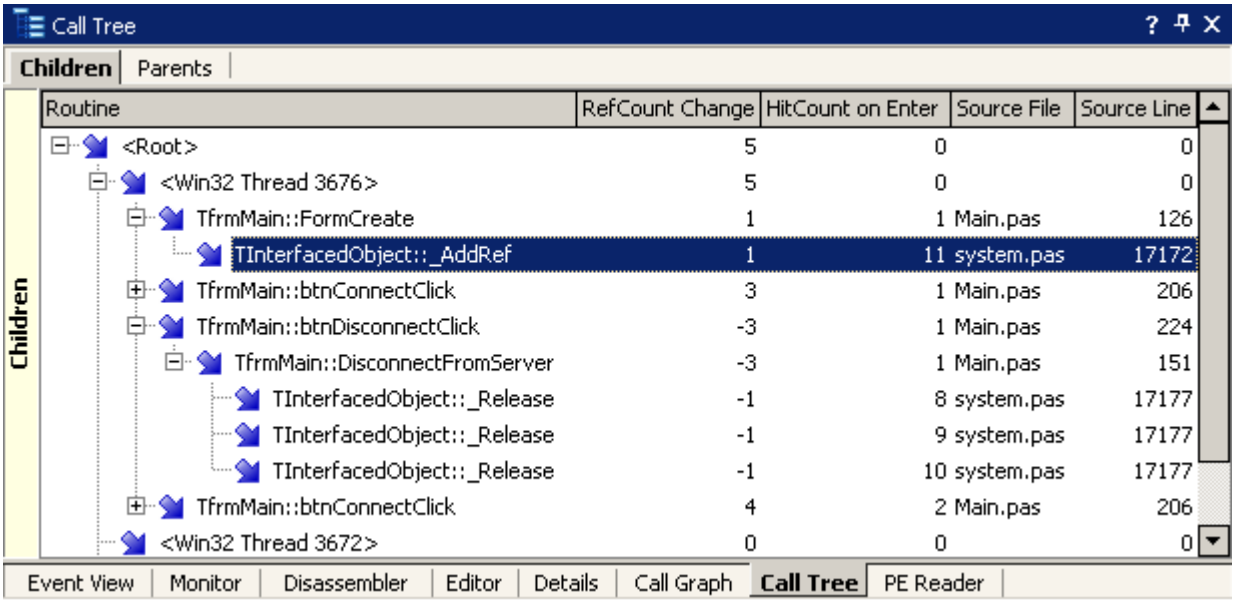
© 2010 AutomatedQA Corp.



AQtime standalone

When the Objects category is active in the Report panel, you can switch to the Details, Call Tree and Call Graph panels in order to find additional information about the selected object. The Details panel includes three panes, **Creation Call Stack**, **References** and **AddRef / Release Call Stack**. The Creation Call Stack pane describes how the chosen object was created. The latter two panes report how the object references were made and removed. Both of the call stack panes display the code instructions, and a double-click on any instruction will take you to the corresponding line in the source code (if it is available to the Editor panel).

The Call Graph and Call Tree panels show how the reference counter was modified during the application execution, how it was altered by calls to this or that routine. The Call Graph displays this data in graphical form, while the Call Tree uses the table view with expandable rows.



Like any other profilers, the results are automatically generated when the profiled application is terminated, or can be obtained at runtime via the Get Results command. See, *Getting Results During Testing*. The first way reveals the leaked interface objects, that is objects that still have unreleased references to them. If the results were retrieved after the application was terminated and the Report panel contains any items in the Objects category, then these objects were not properly disposed. The latter way to obtain results allows you to inspect how the references are created/released during the application execution.

Reference Count Profiler - Report Panel Columns

When displaying results of the Reference Count profiler, each row in the Report panel holds the results either for the entire class or for individual objects. Which values are displayed depends on the category selected in the Explorer panel –

- *Classes Data Category*
- *Objects Category*

Classes Data Category

When this category is active, the Report panel shows information about interface classes created during the application runtime. All classes that implement the `IUnknown` interface are listed in the Report panel, even if all class instances were destroyed at the moment the results are generated. The panel holds the following columns:

Columns (in alphabetical order)	**Description**
Class Name	The name of the class.
Live Count	The number of objects that currently exist in memory.
Live Size	The size of the currently existing objects (in bytes).
Module Name	The name of the module in which the class is defined.
Peak Created	The maximum number of concurrent objects reached during the run.
Peak Size	The maximum size of concurrent objects reached during the run (in bytes).
Total Created	The total number of objects that were created during the application run.

© 2010 AutomatedQA Corp.

Total Size	Memory needed for all the objects that were created during the run.

Objects Category

When this category is active, the Report panel shows information about object instances that exist at the moment the results are generated. This means, that if an object was created and destroyed before the results were generated then it is not shown in the Report panel. The panel holds the following columns:

Columns (in alphabetical order)	**Description**
#	The creation number of the given object.
Address	The memory address where the object was allocated.
Get #	The ordinal number of the Get Results command that generated the current result set. For instance, if you pressed Get Results two times during the profiler run, you will get three result sets (the third result set will be generated after the application closes) with numbers 1, 2 and 3. The Get # value in all records of the first result set will hold 1; in the second result set this column will hold 2 and in the third result set the column will hold 3. The Get # column is used for comparison purposes. It lets you easily see which objects were created or deleted between two result generations.
Module Name	The name of the executable module in which the object's class is defined.
Object Name	The name of the object. It is formed as *Class Name* + *period* + *number*. For example, *TTestClass.3* means the third `TTestClass` instance that was created after the profiling started.
Class Name	The name of the object's class.
Size	The size of the object in bytes.
Thread	Specifies the thread in which the object's constructor was called.

Reference Count Profiler - Columns of the Details and Call Tree Panels

The Reference Count profiler organizes results into two categories: **Classes** and **Objects**. When the Objects category is selected in the Explorer panel, the Report panel holds results for each interface object that existed when the results were being generated. If you select a Report panel row that corresponds to a single object instance, the Reference Count profiler will display information about this instance in the Details, Call Tree and Call Graph panels. These three panels are useful only when the Objects category is selected, they are not availabe for the Classes category.

The Details panel holds three panes: **Creation Call Stack**, **References** and **AddRef / Release Call Stack**. The latter two panes display linked data and by default are shown in a single layout.

The **Creation Call Stack** pane displays the routine calls that led to the creation of the object instance selected in the Report panel. The routine that created the given instance is shown at the top of the call stack. The pane has the following columns:

Columns (in alphabetical order)	**Description**
Class Name	The name of the object class that holds the routine.
Hit Count	The number of routine calls per selected routine.
Module Name	The name of the module that holds the routine.
Routine Name	The name of the routine.

Source File	The name of the source file for the routine. The values for this column are read from the application's debug info. If the debug info does not contain information on the file name, the column is empty.
Source Line	The source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.

The **References** pane contains a single column listing the methods that increased (`AddRef`) or decreased (`Release`) the reference counter of the object selected in the Report panel. The sequence of routines that caused a certain method call is shown in the **AddRef / Release Call Stack** pane. These two panes complement each other. The AddRef / Release Call Stack pane has the following columns:

Columns (in alphabetical order)	**Description**
Call No	The caller rank in the call stack. The topmost caller has an index of 0.
Module Name	The name of the module that holds the routine.
Routine Name	The name of the routine.
Source File	The name of the source file for the routine. The values for this column are read from the application's debug info. If the debug info does not contain information on the file name, the column is empty.
Source Line	The source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.

You can double-click any routine in the **Creation Call Stack** pane or in the **AddRef / Release Call Stack** table to open (if the source file is available) the corresponding code line in the Editor panel.

The **Call Tree** and **Call Graph** panels show changes made to the reference counter during the application run. They both represent the hierarchy of the routines used to create or delete references: the Call Graph provides a graphical scheme, whereas the Call Tree panel provides information in tables. The Call Tree panel contains the following columns:

Columns (in alphabetical order)	**Description**
Hit Count on Enter	The routines' call number that signifies when it was placed in the stack.
RefCount Change	The number by which the reference counter was modified.
Routine	The name of the routine.
Source File	The name of the source file for the routine. The values for this column are read from the application's debug info. If the debug info does not contain information on the file name, the column is empty.
Source Line	The source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.

Reference Count Profiler Options

The Reference Count profiler includes a number of options that can be customized. To modify them do any of the following:

- *AQtime standalone:*
  - select **Options | Options** from AQtime's main menu and then choose **Profilers | Allocation | Reference Count Profiler** from the tree view on the left of the Options dialog;
  - press **Configure Current Profiler** on the **Standard** toolbar when the **Reference Count** profiler is selected.

- *AQtime integrated into Visual Studio:*
  - select **Tools | Options** from Visual Studio's main menu and then choose **AQtime | Profilers | Allocation | Reference Count Profiler** from the tree view on the left of the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - select **Profile | Options** from Borland Developer Studio's main menu and then choose **Profilers | Allocation | Reference Count Profiler** from the tree view on the left of the Options dialog.

In the dialog, the following options are available:

- **Collect stack information** - Specifies how the profiler should collect information on call stacks when creating objects. The following values are available: *None*, *By routines* and *By lines*. Tracing the call stack can significantly slow down the profiled application. If you do not need information on call stacks, you can set this option to *None*. *By routines* means that call stack entries will only include information about routines. If you want the call stack entries to include information on source line numbers as well, set the option to *By lines*. This will let you, for example, determine from which source line a function was called. Tracing source lines, however, requires time.
- **Thread model** - Specifies how the Reference Count profiler gathers statistics for threads in the profiled application. For more information on available values for this option, see *Profiling Multiple Threads* and *Profiling COM Logical Threads*.

Platform Compliance Profiler

Platform Compliance Profiler - Overview

This topic provides the Platform Compliance profiler overview and describes the profiler results. The complete profiler description includes the following topics:

- *Overview (this topic)*
- *Description of the Report Panel Columns*
- *Platform Compliance Profiler Options*

Overview

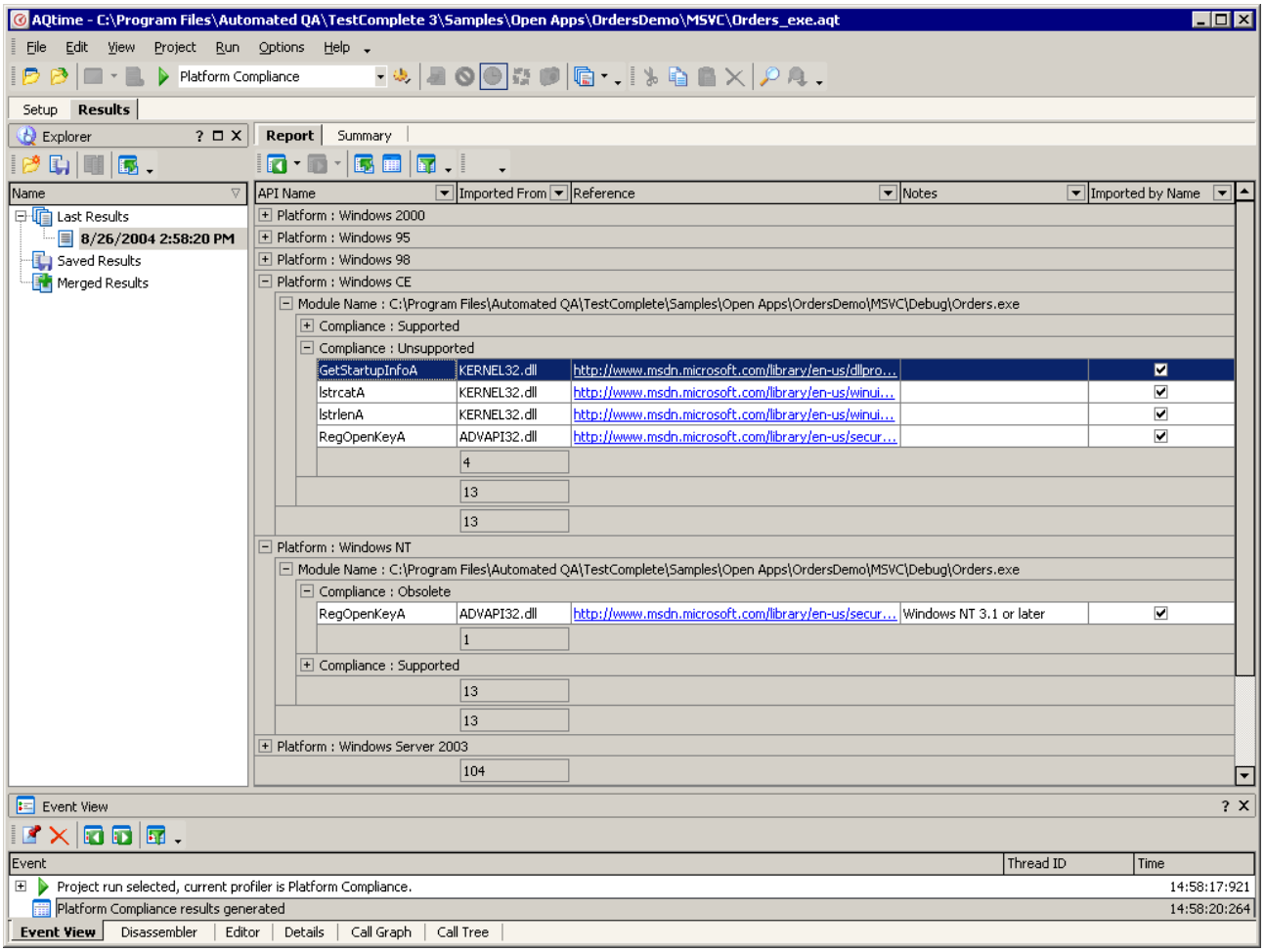
The **Platform Compliance** profiler helps determine whether the profiled application can work on a specific operating system. It is a static profiler. Running it means running the profiler on the application, but not running the application itself.

The Platform Compliance profiler analyzes unmanaged modules only. It is so remarkably informative, complete and easy to run (a matter of seconds) that you will likely run it on all your applications. Unlike other profilers it does not require the application to be compiled with the debugger information. Its one limitation is that it depends on static information. All platform calls under Windows are DLL calls. DLL's can be statically linked, that is, be called using addresses defined at compile time, or dynamically linked, that is, be called using addresses found at runtime only, and especially from DLL's that are likely to change. The Platform Compliance cannot check dynamically defined calls.

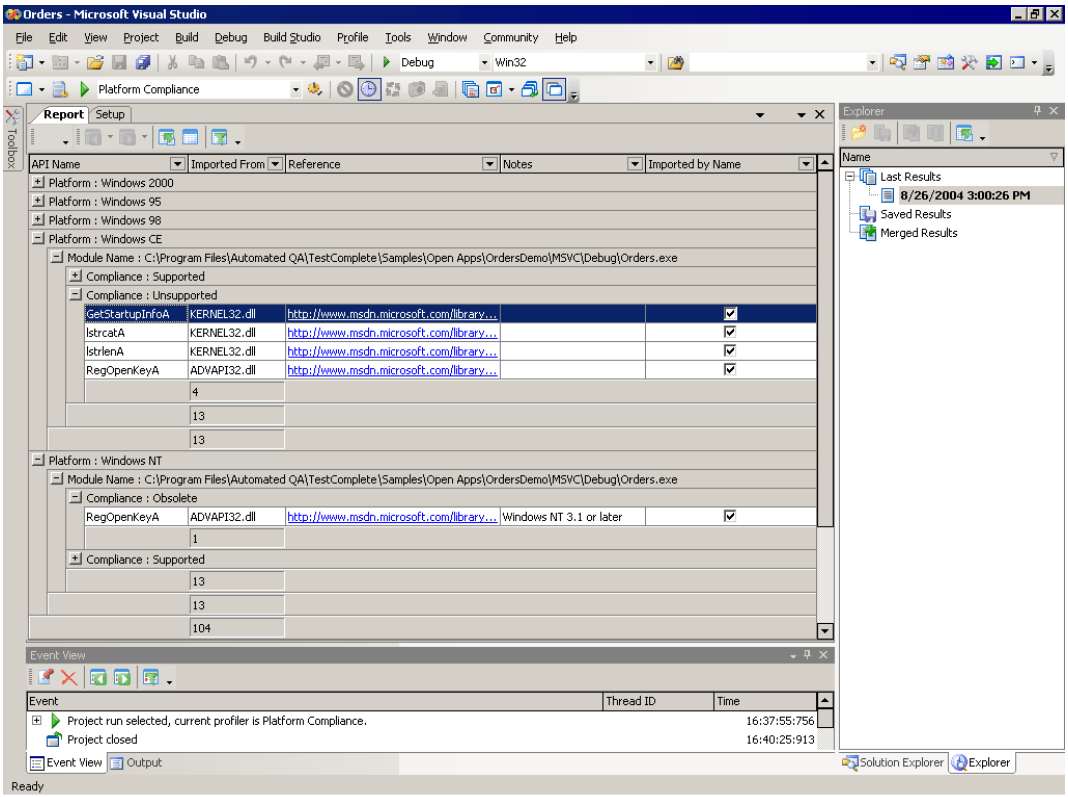
This limitation especially affects the profiling of Visual Basic applications. In Visual Basic, statically linked calls are those defined through the `DECLARE` statement. If your VB application uses only the MSVBVM50 (Visual Basic 5.0) or MSVBVM60 (Visual Basic 6.0) libraries and uses no `DECLARE` of its own, the Platform Compliance profiler will yield no information at all in the **Report** panel.

Statically linked calls are said to be exported, and their target functions to be imported at runtime from system libraries which export them (make them available to external calls). The Platform Compliance profiler analyzes all of the exported calls that address system libraries (that is, DLLs), and checks which of the platform libraries will support the call and how. This analysis is done against a database of compliance information that is part of the AQtime installation.

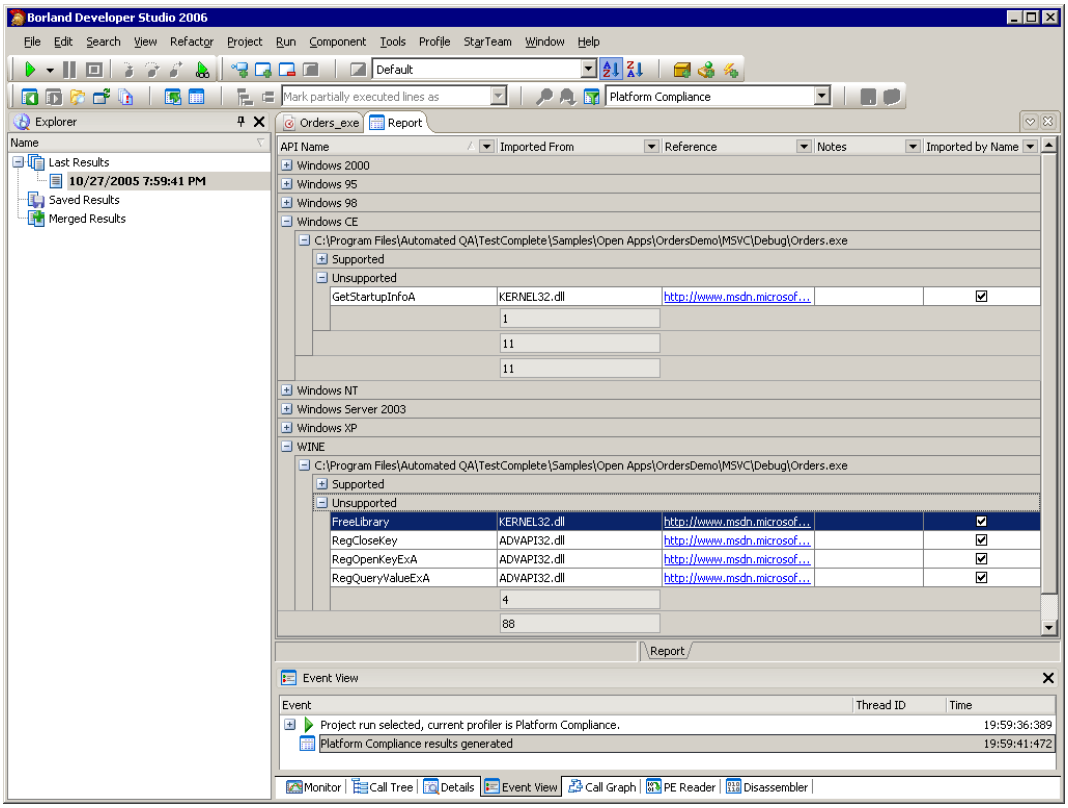
Here is an example output of the Platform Compliance profiler:



AQtime standalone



AQtime integrated into Visual Studio



AQtime integrated into Borland Developer Studio

As you can see, each row in the Report panel holds a compliance result for an operating system routine that is used by your application. By default, the profiler results are grouped by the **Platform**, **Compliance** and **Module Name** columns. For complete information on the Report panel columns, see *Platform Compliance Profiler - Report Panel Columns*.

The Platform Compliance profiler uses a special type of filtering that is set before the data collection. Calls with a certain compliance status can be eliminated from the results for clarity. This pre-filtering is set by the **Platform Compliance Settings** dialog. If the *Show Platform Compliance Settings* option is enabled, the dialog will be displayed before starting each run of the Platform Compliance profiler.

Warning-type compliance values are always included in the results. They are – *Non-Functional*, *Special Usage*, *Special Requirements* and *Unknown*. Other compliance categories can be filtered out through the first two settings in the dialog:

Setting	Description
Obsolete and Unsupported	Supported functions are removed from results. Functions of other categories remain.
Only Unsupported	Supported and Obsolete functions are removed from results. Functions of other categories remain.
Full Analysis	All compliance categories are included in results.

Platform Compliance Profiler - Report Panel Columns

When displaying results of the **Platform Compliance** profiler, the **Report** panel holds the following columns:

Columns (in alphabetical order)	**Description**
API Name	Function name according to the system API (Application Programming Interface).
Compliance	Compliance of the call for the platform specified by the **Platform** column. This column can display the following values: • *Supported* – The call is correctly supported. • *Unsupported* – The function is absent from the DLL for this OS. When loading the DLL, the application will display an error message. • *Obsolete* – The function is present and active, but obsolete. Using it is not recommended by the platform maker. • *Special Requirements* – The call will be supported if some additional software is present on the machine, as listed in the **Notes** column. If the software is absent, the compliance status is *Unsupported*. For instance, some calls to Windows NT 4.0 require Service Pack 4.0 or later. • *Non-functional* - Though MSDN reports this function as "unsupported", it is present and will prevent an error message from being posted. However, calls to it will do nothing. • *Special Usage* - The function is present and active, but it is not fully functional. E.g. it may ignore some parameters. • *Unknown* - AQtime does not have any information about the function.

Imported by Name	If this column is checked, the routine is imported by name. Otherwise, the routine is imported by ordinal.
Imported From	Name of the system dynamic link library which exports the function. This name may or may not have the *dll* extension.
Module Name	Name of the application module (.exe, .dll, etc.) which calls the function.
Notes	Additional information about compliance conditions.
Platform	Name of the platform (operating system).
Reference	Hyperlink to the on-line function documentation. To open it, simply double-click on the cell.

Platform Compliance Profiler Options

The **Platform Compliance** profiler offers the only customizable option - *Compliance level*. It specifies what type of API calls to check for. It can be one of the following values:

Value	Description
Obsolete and Unsupported	Supported functions are removed from results. Functions of other categories (special requirements, obsolete, non-functional, etc.) remain.
Only Unsupported	Supported and Obsolete functions are removed from results. Functions of other categories remain.
Full Analysis	All compliance categories are included in results.

To change the option, do any of the following:

- *AQtime standalone:*
  - select **Options | Options** from AQtime's main menu and then choose **Profilers | Static Analysis | Platform Compliance** in the ensuing Options dialog;
  - press **Configure Current Profiler** on the **Standard** toolbar when the **Platform Compliance** profiler is selected.
- *AQtime integrated into Visual Studio:*
  - select **Tools | Options** from Visual Studio's main menu and then choose the **AQtime | Profilers | Static Analysis | Platform Compliance** group in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - select **Profile | Options** from Borland Developer Studio's main menu and then choose **Profilers | Static Analysis | Platform Compliance** in the ensuing Options dialog.

Resource Profiler

Resource Profiler - Overview

The **Resource** profiler checks to see if the application being profiled creates Windows resources correctly and releases all of the allocated resources. (Both 32-bit and 64-bit applications are supported.) This topic provides the Resource profiler overview. The complete profiler description includes the following topics:

- *Overview (this topic)*
- *Analyzing Resource Profiler Results*

- *Description of the Report Panel Columns*
- *Description of the Details Panel Columns*
- *Resource Profiler Options*

Overview

The Resource profiler keeps tabs on the Windows resource usage monitoring resource allocations and deallocations and calls to the the resource management routines. For instance, if the application attempts to delete a pen currently selected in the device context, the profiler will add the "Attempt to delete the object selected in DC" error message to the Report panel. For the full list of the checked resource management functions, see *Resource Profiler - List of Checked Functions*.

Note: AQtime may report that there was an attempt to free a non-allocated resource, while this resource actually is allocated. AQtime may also report about non-existent resources if you profile your application using the “Attach to Process” feature. For more information, see *Non-Existent Resources in the Report panel*.

The Resource profiler tracks calls to Windows API functions that deal with resources. In managed (.NET-connected) applications, work with resources is organized via objects that implement the `IDisposable` interface. The Resource profiler does not track allocations and deallocations of these resources.

The Resource profiler traces the resource usage in both managed and unmanaged (native-code) applications. The profiler always traces the entire application to be profiled; it ignores areas and the *Profile Entire .NET Code* item. The Resource profiler operates during the entire run of the application. It takes no account of triggers and actions and disables the **Enable/Disable Profiling** button.

By default, the **Resource** profiler reports only the resource types whose instances had been allocated before the results were generated. However, the profiler includes the *Show all resources* option that lets you extend the report. This option is displayed on the AQtime’s or Borland Developer Studio’s **Profiler** toolbar or Visual Studio’s **Report** toolbar ( ). If this option is enabled, the profiler reports about all the resource types the profiler traces even if no instances were allocated for these resource types. If you do not want to see these resource types in the report, uncheck the *Show all resources* option.

The second option on the same toolbar, **View project classes only**, controls for which modules AQtime displays resource-profiling results. If this option is disabled (this is its default state), AQtime displays profiling results for all the modules used by the application being profiled. Often, this setting substantially extends the report. If the option is enabled, AQtime displays profiling results only for those modules that are added to the Setup panel.

The third toolbar option, **Filter objects by stack**, operates on a similar principle. If this option is enabled, AQtime only displays those objects that were created in one of the “Setup” modules. If the option is disabled (default state) all traced objects are displayed.

The last filtering option is **Filter standard leaks**, it hides information about resource leaks that occured in standard third-party classes and libraries (VCL, MFC and others).

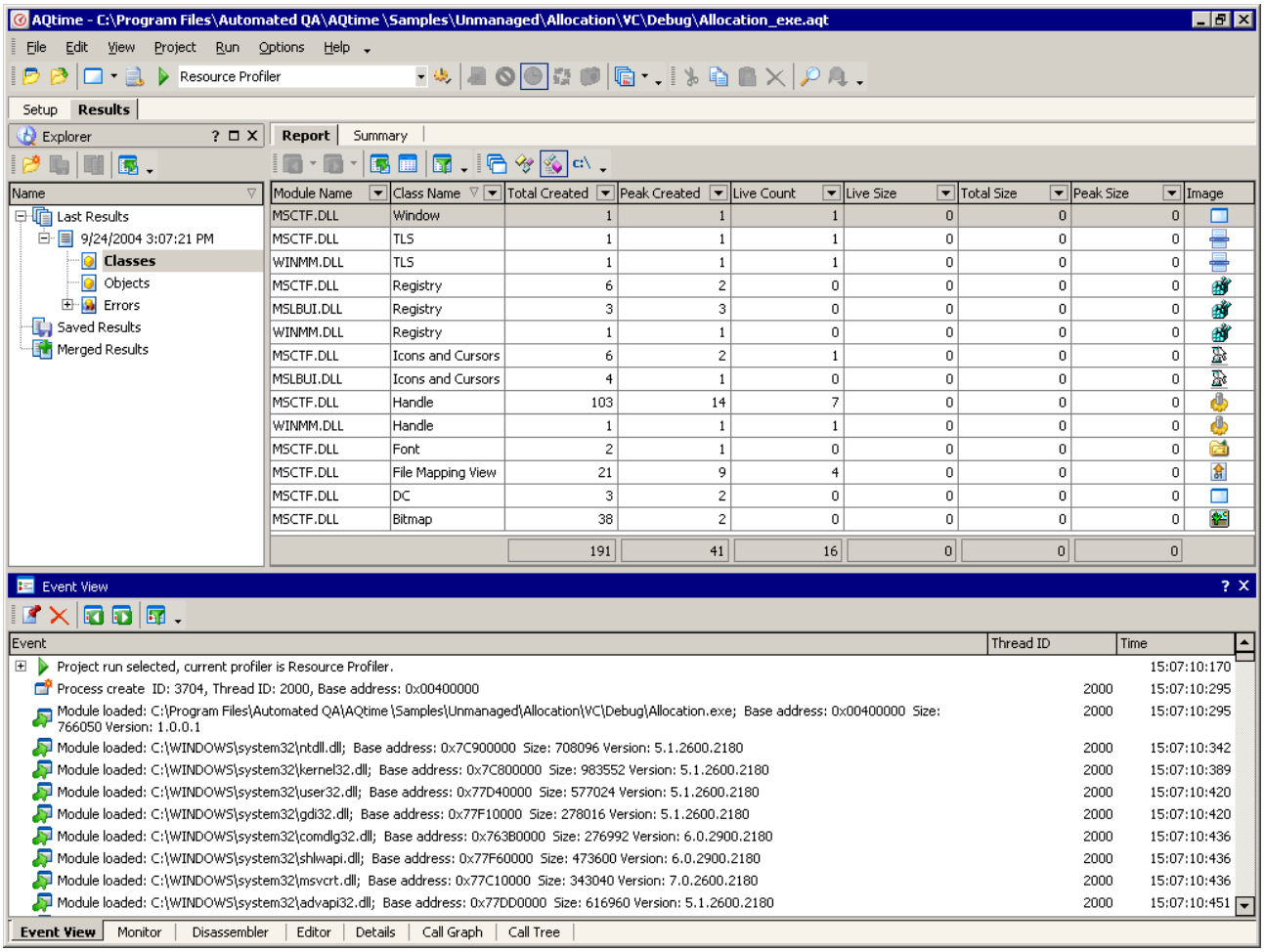
Using these options, you can get rid of profiling results you do not need at the moment.

Analyzing Resource Profiler Results

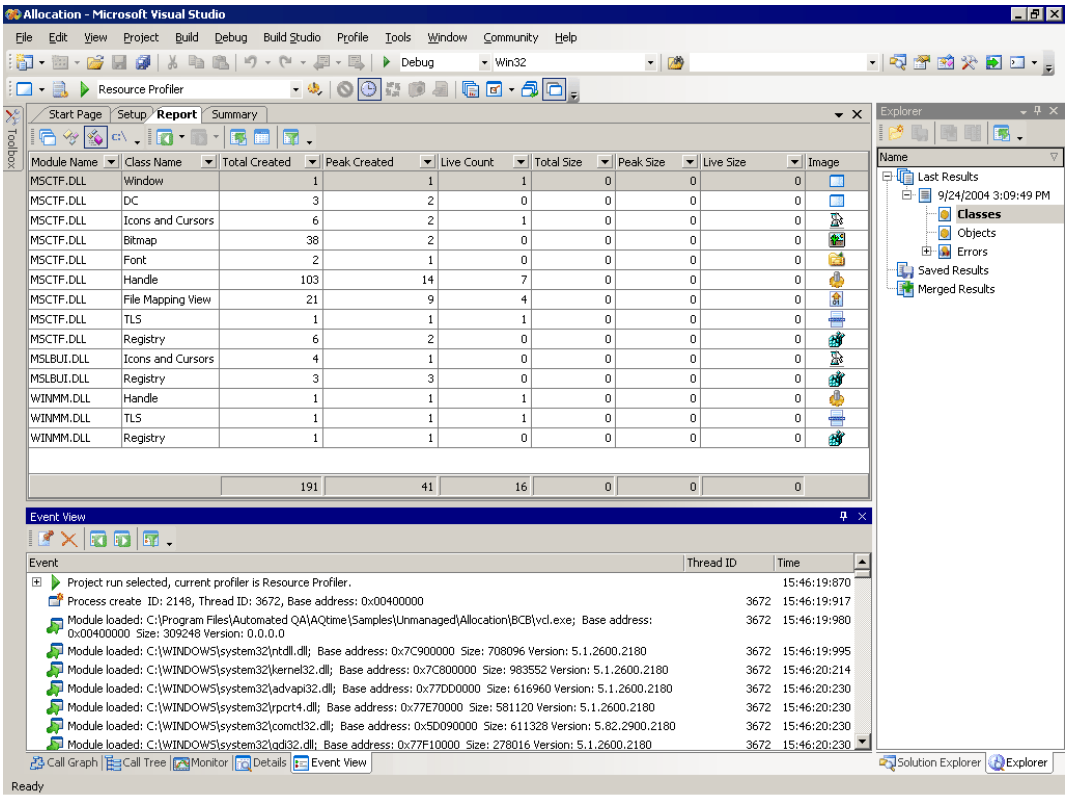
The **Resource** profiler traces how the profiled application uses Windows resources. Like other AQtime profilers, the Resource profiler generates results upon selecting **Run | Get Results** from the main menu of AQtime (if it is running as a standalone application), selecting **Profile | Get Results** from Visual Studio’s

menu (if AQtime is integrated into Visual Studio), clicking **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar (if AQtime is integrated into Borland Developer Studio) or after the application terminates.

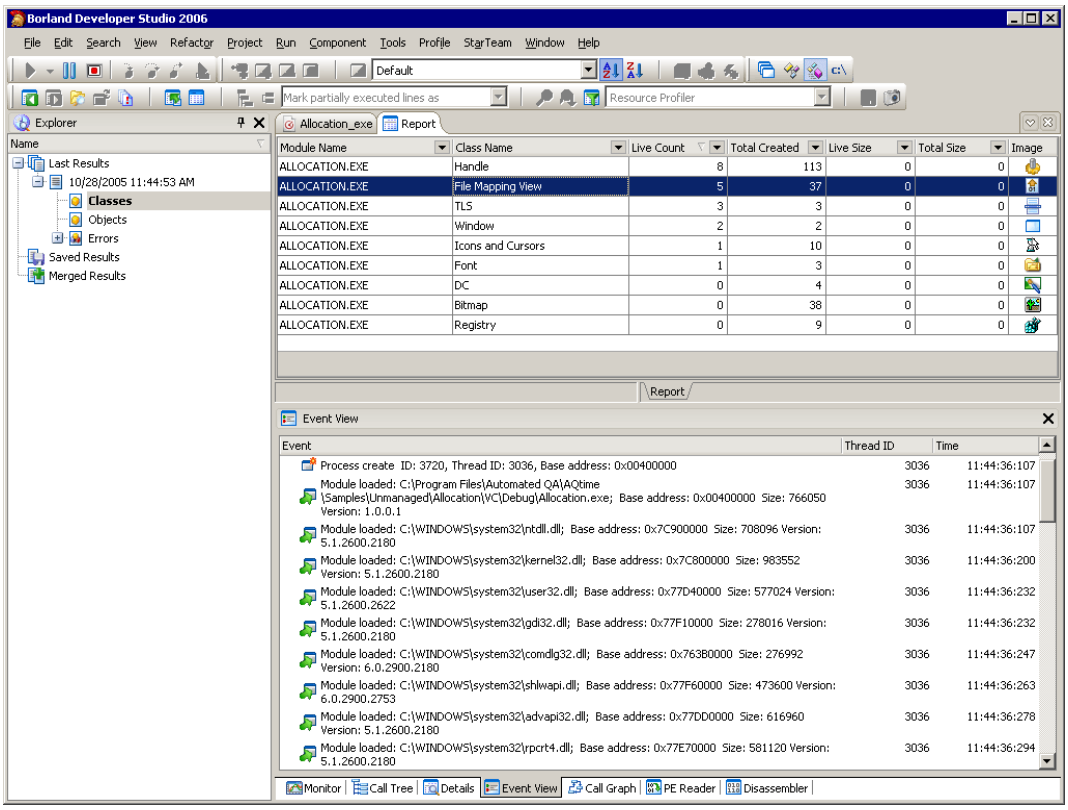
The **Summary** panel displays brief profiling results. It reports about resources with maximum number of existing instances, resources with maximum number of created instances, etc. Information about all the resources that are used by the application is shown in the **Report** panel. Here is a sample output of the Resource profiler:


AQtime - C:\Program Files\Automated QA\AQtime \Samples\Unmanaged\Allocation\VC\Debug\Allocation_exe.aqt
File Edit View Project Run Options Help
Resource Profiler
Setup Results
Explorer
Report Summary
Name
Last Results
9/24/2004 3:07:21 PM
Classes
Objects
Errors
Saved Results
Merged Results
Module Name | Class Name | Total Created | Peak Created | Live Count | Live Size | Total Size | Peak Size | Image
MSCTF.DLL | Window | 1 | 1 | 1 | 0 | 0 | 0
MSCTF.DLL | TLS | 1 | 1 | 1 | 0 | 0 | 0
WINMM.DLL | TLS | 1 | 1 | 1 | 0 | 0 | 0
MSCTF.DLL | Registry | 6 | 2 | 0 | 0 | 0 | 0
MSLBUI.DLL | Registry | 3 | 3 | 0 | 0 | 0 | 0
WINMM.DLL | Registry | 1 | 1 | 0 | 0 | 0 | 0
MSCTF.DLL | Icons and Cursors | 6 | 2 | 1 | 0 | 0 | 0
MSLBUI.DLL | Icons and Cursors | 4 | 1 | 0 | 0 | 0 | 0
MSCTF.DLL | Handle | 103 | 14 | 7 | 0 | 0 | 0
WINMM.DLL | Handle | 1 | 1 | 1 | 0 | 0 | 0
MSCTF.DLL | Font | 2 | 1 | 0 | 0 | 0 | 0
MSCTF.DLL | File Mapping View | 21 | 9 | 4 | 0 | 0 | 0
MSCTF.DLL | DC | 3 | 2 | 0 | 0 | 0 | 0
MSCTF.DLL | Bitmap | 38 | 2 | 0 | 0 | 0 | 0
191 41 16 0 0 0
Event View
Event | Thread ID | Time
Project run selected, current profiler is Resource Profiler. 15:07:10:170
Process create ID: 3704, Thread ID: 2000, Base address: 0x00400000 2000 15:07:10:295
Module loaded: C:\Program Files\Automated QA\AQtime \Samples\Unmanaged\Allocation\VC\Debug\Allocation.exe; Base address: 0x00400000 Size: 766050 Version: 1.0.0.1 2000 15:07:10:295
Module loaded: C:\WINDOWS\system32\ntdll.dll; Base address: 0x7C900000 Size: 708096 Version: 5.1.2600.2180 2000 15:07:10:342
Module loaded: C:\WINDOWS\system32\kernel32.dll; Base address: 0x7C800000 Size: 983552 Version: 5.1.2600.2180 2000 15:07:10:389
Module loaded: C:\WINDOWS\system32\user32.dll; Base address: 0x77D40000 Size: 577024 Version: 5.1.2600.2180 2000 15:07:10:420
Module loaded: C:\WINDOWS\system32\gdi32.dll; Base address: 0x77F10000 Size: 278016 Version: 5.1.2600.2180 2000 15:07:10:420
Module loaded: C:\WINDOWS\system32\comdlg32.dll; Base address: 0x763B0000 Size: 276992 Version: 6.0.2900.2180 2000 15:07:10:436
Module loaded: C:\WINDOWS\system32\shlwapi.dll; Base address: 0x77F60000 Size: 473600 Version: 6.0.2900.2180 2000 15:07:10:436
Module loaded: C:\WINDOWS\system32\msvcrt.dll; Base address: 0x77C10000 Size: 343040 Version: 7.0.2600.2180 2000 15:07:10:436
Module loaded: C:\WINDOWS\system32\advapi32.dll; Base address: 0x77DD0000 Size: 616960 Version: 5.1.2600.2180 2000 15:07:10:451
Event View Monitor Disassembler Editor Details Call Graph Call Tree

AQtime standalone


Allocation - Microsoft Visual Studio
File Edit View Project Build Debug Build Studio Profile Tools Window Community Help
Debug
Win32
Resource Profiler
Start Page Setup Report Summary
Module Name Class Name Total Created Peak Created Live Count Total Size Peak Size Live Size Image
MSCTF.DLL Window 1 1 1 0 0 0
MSCTF.DLL DC 3 2 0 0 0 0
MSCTF.DLL Icons and Cursors 6 2 1 0 0 0
MSCTF.DLL Bitmap 38 2 0 0 0 0
MSCTF.DLL Font 2 1 0 0 0 0
MSCTF.DLL Handle 103 14 7 0 0 0
MSCTF.DLL File Mapping View 21 9 4 0 0 0
MSCTF.DLL TLS 1 1 1 0 0 0
MSCTF.DLL Registry 6 2 0 0 0 0
MSLBUI.DLL Icons and Cursors 4 1 0 0 0 0
MSLBUI.DLL Registry 3 3 0 0 0 0
WINMM.DLL Handle 1 1 1 0 0 0
WINMM.DLL TLS 1 1 1 0 0 0
WINMM.DLL Registry 1 1 0 0 0 0
191 41 16 0 0 0
Event View
Event Thread ID Time
Project run selected, current profiler is Resource Profiler. 15:46:19:870
Process create ID: 2148, Thread ID: 3672, Base address: 0x00400000 3672 15:46:19:917
Module loaded: C:\Program Files\Automated QA\AQtime\Samples\Unmanaged\Allocation\BCB\vcl.exe; Base address: 0x00400000 Size: 309248 Version: 0.0.0.0 3672 15:46:19:980
Module loaded: C:\WINDOWS\system32\ntdll.dll; Base address: 0x7C900000 Size: 708096 Version: 5.1.2600.2180 3672 15:46:19:995
Module loaded: C:\WINDOWS\system32\kernel32.dll; Base address: 0x7C800000 Size: 983552 Version: 5.1.2600.2180 3672 15:46:20:214
Module loaded: C:\WINDOWS\system32\advapi32.dll; Base address: 0x77DD0000 Size: 616960 Version: 5.1.2600.2180 3672 15:46:20:230
Module loaded: C:\WINDOWS\system32\rpcrt4.dll; Base address: 0x77E70000 Size: 581120 Version: 5.1.2600.2180 3672 15:46:20:230
Module loaded: C:\WINDOWS\system32\comctl32.dll; Base address: 0x5D090000 Size: 611328 Version: 5.82.2900.2180 3672 15:46:20:230
Module loaded: C:\WINDOWS\system32\gdi32.dll; Base address: 0x77F10000 Size: 278016 Version: 5.1.2600.2180 3672 15:46:20:230
Call Graph Call Tree Monitor Details Event View
Explorer
Name
Last Results
9/24/2004 3:09:49 PM
Classes
Objects
Errors
Saved Results
Merged Results
Solution Explorer Explorer
Ready

AQtime integrated into Visual Studio

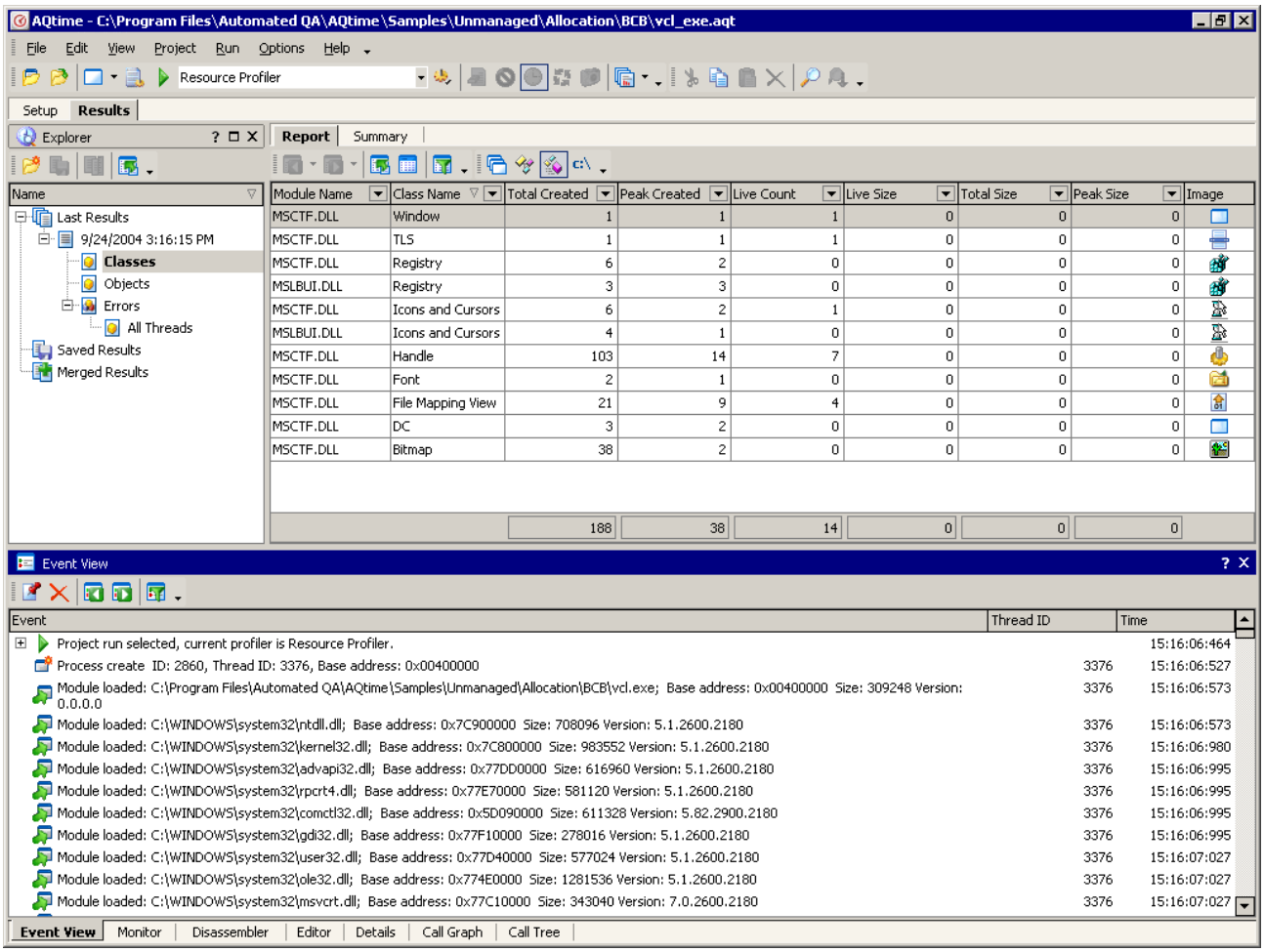

Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Default
Mark partially executed lines as
Resource Profiler
Explorer
Name
Last Results
10/28/2005 11:44:53 AM
Classes
Objects
Errors
Saved Results
Merged Results
Allocation_exe Report
Module Name Class Name Live Count Total Created Live Size Total Size Image
ALLOCATION.EXE Handle 8 113 0 0
ALLOCATION.EXE File Mapping View 5 37 0 0
ALLOCATION.EXE TLS 3 3 0 0
ALLOCATION.EXE Window 2 2 0 0
ALLOCATION.EXE Icons and Cursors 1 10 0 0
ALLOCATION.EXE Font 1 3 0 0
ALLOCATION.EXE DC 0 4 0 0
ALLOCATION.EXE Bitmap 0 38 0 0
ALLOCATION.EXE Registry 0 9 0 0
Report
Event View
Event Thread ID Time
Process create ID: 3720, Thread ID: 3036, Base address: 0x00400000 3036 11:44:36:107
Module loaded: C:\Program Files\Automated QA\AQtime \Samples\Unmanaged\Allocation\VC\Debug\Allocation.exe; Base address: 0x00400000 Size: 766050 Version: 1.0.0.1 3036 11:44:36:107
Module loaded: C:\WINDOWS\system32\ntdll.dll; Base address: 0x7C900000 Size: 708096 Version: 5.1.2600.2180 3036 11:44:36:107
Module loaded: C:\WINDOWS\system32\kernel32.dll; Base address: 0x7C800000 Size: 983552 Version: 5.1.2600.2180 3036 11:44:36:200
Module loaded: C:\WINDOWS\system32\user32.dll; Base address: 0x77D40000 Size: 577024 Version: 5.1.2600.2622 3036 11:44:36:232
Module loaded: C:\WINDOWS\system32\gdi32.dll; Base address: 0x77F10000 Size: 278016 Version: 5.1.2600.2180 3036 11:44:36:232
Module loaded: C:\WINDOWS\system32\comdlg32.dll; Base address: 0x763B0000 Size: 276992 Version: 6.0.2900.2180 3036 11:44:36:247
Module loaded: C:\WINDOWS\system32\shlwapi.dll; Base address: 0x77F60000 Size: 473600 Version: 6.0.2900.2753 3036 11:44:36:263
Module loaded: C:\WINDOWS\system32\advapi32.dll; Base address: 0x77DD0000 Size: 616960 Version: 5.1.2600.2180 3036 11:44:36:278
Module loaded: C:\WINDOWS\system32\rpcrt4.dll; Base address: 0x77E70000 Size: 581120 Version: 5.1.2600.2180 3036 11:44:36:294
Monitor Call Tree Details Event View Call Graph PE Reader Disassembler

AQtime integrated into Borland Developer Studio

As you can see, the Resource profiler results are divided into three categories: **Classes**, **Objects** and **Errors**. These categories are displayed as subnodes of the result set node in the **Explorer** panel. The contents of the other panels (**Report**, **Details**, **Editor**, etc.) depend on the currently selected category.

- **Classes**. When this category is selected, the Report panel displays information about resource types whose instances were created during the run.

  Each row in the Report panel shows profiling results for every single resource type whose instances were allocated from the given module (for instance, if the profiled application created resources of the Handle type from two modules, e.g. msctf.dll and winmm.dll, the Report panel will have two records about this resource type, one for each module). These profiling results include the total and current number of resource instances, their size, etc. For more detailed information, review *Resource Profiler - Report Panel Columns*. This gives you a summary view on what happened with resources in the application during profiling (you can also view the summary results in the Summary panel). Note that by default the Report panel holds only some of available columns. You can add more columns to the panel or remove columns from it. For more information on this, see *Adding and Removing Columns*.


AQtime - C:\Program Files\Automated QA\AQtime\Samples\Unmanaged\Allocation\BCB\vcl_exe.aqt
File Edit View Project Run Options Help
Resource Profiler
Setup Results
Explorer
Report Summary
Name
Last Results
9/24/2004 3:16:15 PM
Classes
Objects
Errors
All Threads
Saved Results
Merged Results
Module Name | Class Name | Total Created | Peak Created | Live Count | Live Size | Total Size | Peak Size | Image
MSCTF.DLL | Window | 1 | 1 | 1 | 0 | 0 | 0
MSCTF.DLL | TLS | 1 | 1 | 1 | 0 | 0 | 0
MSCTF.DLL | Registry | 6 | 2 | 0 | 0 | 0 | 0
MSLBUI.DLL | Registry | 3 | 3 | 0 | 0 | 0 | 0
MSCTF.DLL | Icons and Cursors | 6 | 2 | 1 | 0 | 0 | 0
MSLBUI.DLL | Icons and Cursors | 4 | 1 | 0 | 0 | 0 | 0
MSCTF.DLL | Handle | 103 | 14 | 7 | 0 | 0 | 0
MSCTF.DLL | Font | 2 | 1 | 0 | 0 | 0 | 0
MSCTF.DLL | File Mapping View | 21 | 9 | 4 | 0 | 0 | 0
MSCTF.DLL | DC | 3 | 2 | 0 | 0 | 0 | 0
MSCTF.DLL | Bitmap | 38 | 2 | 0 | 0 | 0 | 0
188 38 14 0 0 0
Event View
Event | Thread ID | Time
Project run selected, current profiler is Resource Profiler. 15:16:06:464
Process create ID: 2860, Thread ID: 3376, Base address: 0x00400000 3376 15:16:06:527
Module loaded: C:\Program Files\Automated QA\AQtime\Samples\Unmanaged\Allocation\BCB\vcl.exe; Base address: 0x00400000 Size: 309248 Version: 0.0.0.0 3376 15:16:06:573
Module loaded: C:\WINDOWS\system32\ntdll.dll; Base address: 0x7C900000 Size: 708096 Version: 5.1.2600.2180 3376 15:16:06:573
Module loaded: C:\WINDOWS\system32\kernel32.dll; Base address: 0x7C800000 Size: 983552 Version: 5.1.2600.2180 3376 15:16:06:980
Module loaded: C:\WINDOWS\system32\advapi32.dll; Base address: 0x77DD0000 Size: 616960 Version: 5.1.2600.2180 3376 15:16:06:995
Module loaded: C:\WINDOWS\system32\rpcrt4.dll; Base address: 0x77E70000 Size: 581120 Version: 5.1.2600.2180 3376 15:16:06:995
Module loaded: C:\WINDOWS\system32\comctl32.dll; Base address: 0x5D090000 Size: 611328 Version: 5.82.2900.2180 3376 15:16:06:995
Module loaded: C:\WINDOWS\system32\gdi32.dll; Base address: 0x77F10000 Size: 278016 Version: 5.1.2600.2180 3376 15:16:06:995
Module loaded: C:\WINDOWS\system32\user32.dll; Base address: 0x77D40000 Size: 577024 Version: 5.1.2600.2180 3376 15:16:07:027
Module loaded: C:\WINDOWS\system32\ole32.dll; Base address: 0x774E0000 Size: 1281536 Version: 5.1.2600.2180 3376 15:16:07:027
Module loaded: C:\WINDOWS\system32\msvcrt.dll; Base address: 0x77C10000 Size: 343040 Version: 7.0.2600.2180 3376 15:16:07:027
Event View Monitor Disassembler Editor Details Call Graph Call Tree

To determine if resource instances were existing at the moment of results generation, check the **Live Count** column value. If it is greater than 0, resource instances existed. If you obtain results upon closing the application, non-zero values in the Live Count column help you find resource leaks. To find them faster, you can sort or filter results on the Live Count column.

Note that if the Resource profiler encounters a call to a function that reallocates a resource that is already allocated (e.g. `CoTaskMemRealloc`, `SysReAllocString`, `SysReAllocStringLen`, etc.), the profiler "thinks" that this function deallocates the existing resource and allocates it anew (this is what really happens to this resource). Thus, the profiler increments the total number of resource instances of the corresponding type that were created during the run. For instance, if you call the `SysAllocString` function to allocate a string, the profiler will inform you about one live resource instance of the **Sys strings** type. In this case, the total number of created resource instances of that type will be 1. Then, if you call `SysReAllocString` to reallocate that string, the Resource profiler will inform you that you still have one live resource instance of the Sys strings type, but the total number of created resource instances of that type will be 2.

The footer of the Report panel column holds summary values for data displayed in that column. For instance, the footer of the **Live Size** column displays the summary size of all resource instances that existed at the moment of results generation. If you select two or more resource type records in the Report panel, the footer will show the summary values for the selected resource types only (for more information on how to select several rows in a panel, see *Selecting Several Records in a Panel*).

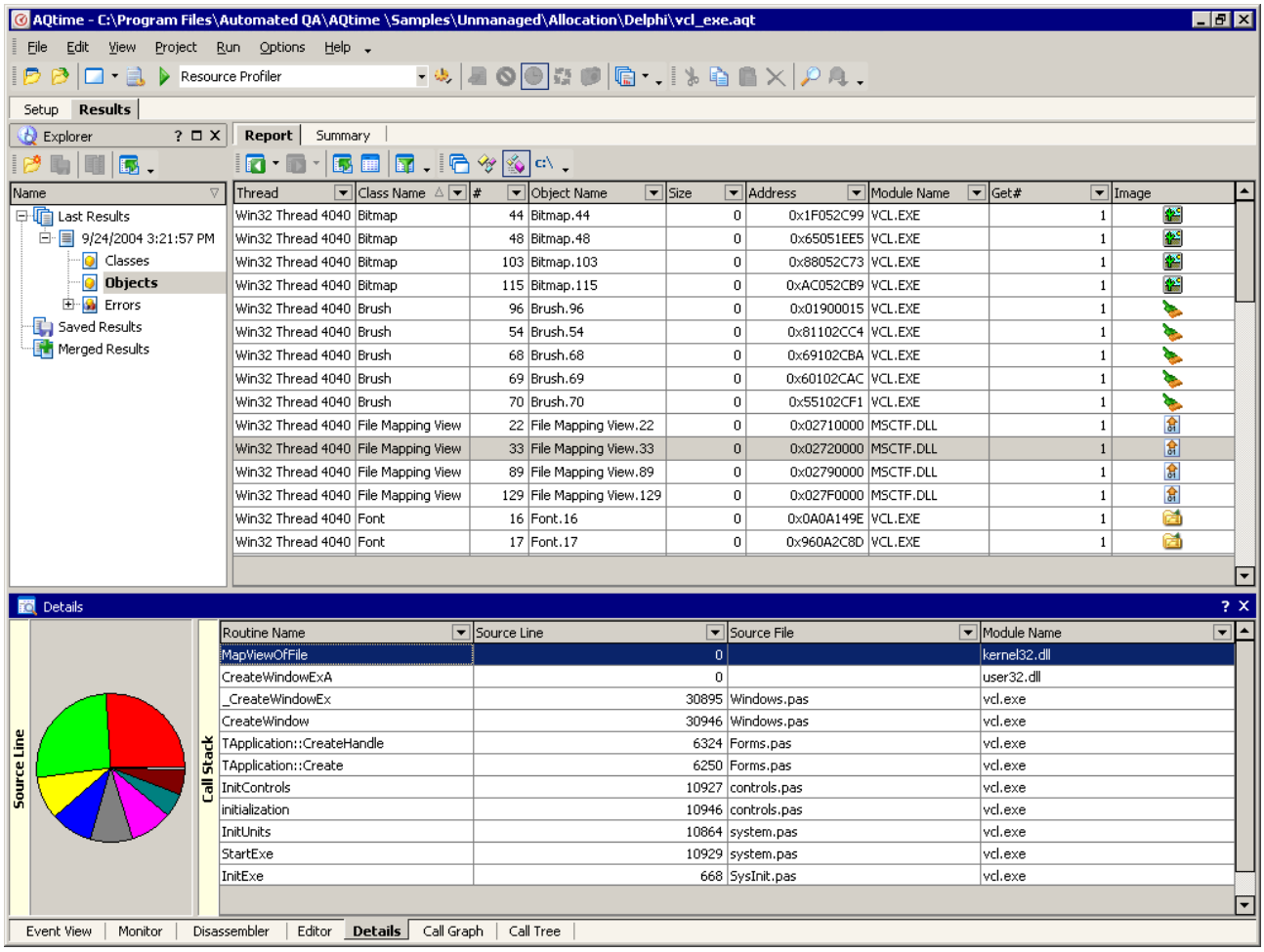
- **Objects**. When this category is selected, the Report panel displays information about resource instances that exist in the application at the moment the results are generated. Every row in the panel holds results for a single resource instance. The **Object Name** column serves as the resource instance identifier. For instance, the name *Icon.5* means the fifth resource instance of the Icon type created after profiling started. To view all resource instances of a certain type, filter results on the **Class Name** column (See *Filtering Results*).

The Report panel columns are completely described in a separate topic, *Resource Profiler - Report Panel Columns*. Here we would like you to pay attention to the **Get #** column. It displays the ordinal number of the result set within a run. For instance, if you pressed **Get Results** during the Resource profiler run, you get two result sets: one that was generated upon pressing that button and another one that were generated upon closing the profiled application. In the first result set, in all records the Get # column will hold 1; in the second result set this column will hold 2. You can use these values for comparison purposes. For instance, when you compare two result sets, the column will clearly tell you what resource instances were created or deleted between the two moments of results generation.

The **Report** panel is the “main” results display for resource instances. The **Details** and **Editor** panels display additional results for the resource instance selected in the Report panel.

The Details panel holds one pane: **Creation Call Stack**. It displays the stack of function calls that led to the resource instance creation. The topmost routine in this stack is the API function that created the resource instance. Columns of the Creation Call Stack page hold information that helps you locate the routine in source code. For more detailed information, review *Resource Profiler - Details Panel Columns*.

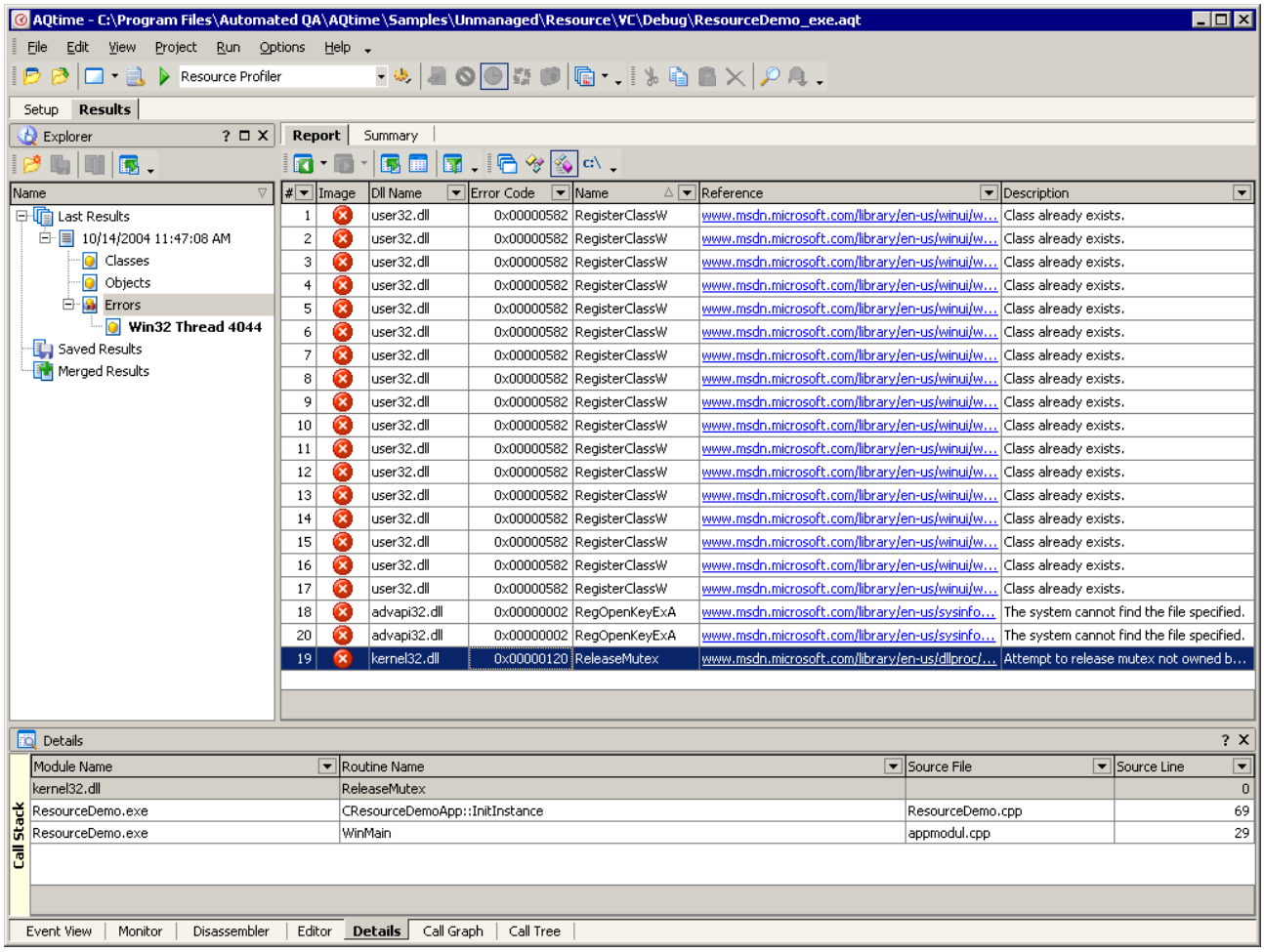
© 2010 AutomatedQA Corp.


AQtime - C:\Program Files\Automated QA\AQtime \Samples\Unmanaged\Allocation\Delphi\vcl_exe.aqt
File Edit View Project Run Options Help
Resource Profiler
Setup Results
Explorer
Report Summary
Name
Last Results
9/24/2004 3:21:57 PM
Classes
Objects
Errors
Saved Results
Merged Results
Thread Class Name # Object Name Size Address Module Name Get# Image
Win32 Thread 4040 Bitmap 44 Bitmap.44 0 0x1F052C99 VCL.EXE 1
Win32 Thread 4040 Bitmap 48 Bitmap.48 0 0x65051EE5 VCL.EXE 1
Win32 Thread 4040 Bitmap 103 Bitmap.103 0 0x88052C73 VCL.EXE 1
Win32 Thread 4040 Bitmap 115 Bitmap.115 0 0xAC052CB9 VCL.EXE 1
Win32 Thread 4040 Brush 96 Brush.96 0 0x01900015 VCL.EXE 1
Win32 Thread 4040 Brush 54 Brush.54 0 0x81102CC4 VCL.EXE 1
Win32 Thread 4040 Brush 68 Brush.68 0 0x69102CBA VCL.EXE 1
Win32 Thread 4040 Brush 69 Brush.69 0 0x60102CAC VCL.EXE 1
Win32 Thread 4040 Brush 70 Brush.70 0 0x55102CF1 VCL.EXE 1
Win32 Thread 4040 File Mapping View 22 File Mapping View.22 0 0x02710000 MSCTF.DLL 1
Win32 Thread 4040 File Mapping View 33 File Mapping View.33 0 0x02720000 MSCTF.DLL 1
Win32 Thread 4040 File Mapping View 89 File Mapping View.89 0 0x02790000 MSCTF.DLL 1
Win32 Thread 4040 File Mapping View 129 File Mapping View.129 0 0x027F0000 MSCTF.DLL 1
Win32 Thread 4040 Font 16 Font.16 0 0x0A0A149E VCL.EXE 1
Win32 Thread 4040 Font 17 Font.17 0 0x960A2C8D VCL.EXE 1
Details
Source Line
Call Stack
Routine Name Source Line Source File Module Name
MapViewOfFile 0 kernel32.dll
CreateWindowExA 0 user32.dll
_CreateWindowEx 30895 Windows.pas vcl.exe
CreateWindow 30946 Windows.pas vcl.exe
TApplication::CreateHandle 6324 Forms.pas vcl.exe
TApplication::Create 6250 Forms.pas vcl.exe
InitControls 10927 controls.pas vcl.exe
initialization 10946 controls.pas vcl.exe
InitUnits 10864 system.pas vcl.exe
StartExe 10929 system.pas vcl.exe
InitExe 668 SysInit.pas vcl.exe
Event View Monitor Disassembler Editor Details Call Graph Call Tree

To view the source code of a routine, simply double-click it in the call stack - AQtime will bring up the **Editor** panel and position the cursor on the first line of the routine's source code (the source file of the routine must be specified in the **Project Search Directories**. In addition, to view sources of your managed applications in the Editor, you should compile the application with debug information. See *How AQtime Profilers Use Metadata and Debug Information*).

The Creation Call Stack is available if the profiler's *Collect stack information option* is set to *By routines* or *By lines*. To disable the call stack tracing, set this option to *None*.

- **Errors**. When this category is selected, the Report panel displays information about errors that occurred in resource-related Windows functions called during the application run. For instance, if the `CreateIcon` function fails, it will be reported here. For each error listed, you can see in which function it occurred, a description of the error and a link to the MSDN topic that holds comprehensive information about the given function. Thus, you can quickly find out what resource-related functions failed (if any) and why this happened.


AQtime - C:\Program Files\Automated QA\AQtime\Samples\Unmanaged\Resource\VC\Debug\ResourceDemo_exe.aqt
File Edit View Project Run Options Help
Resource Profiler
Setup Results
Explorer
Report Summary
Name
Last Results
10/14/2004 11:47:08 AM
Classes
Objects
Errors
Win32 Thread 4044
Saved Results
Merged Results
Image Dll Name Error Code Name Reference Description
1 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
2 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
3 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
4 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
5 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
6 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
7 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
8 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
9 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
10 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
11 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
12 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
13 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
14 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
15 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
16 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
17 user32.dll 0x00000582 RegisterClassW www.msdn.microsoft.com/library/en-us/winui/w... Class already exists.
18 advapi32.dll 0x00000002 RegOpenKeyExA www.msdn.microsoft.com/library/en-us/sysinfo... The system cannot find the file specified.
20 advapi32.dll 0x00000002 RegOpenKeyExA www.msdn.microsoft.com/library/en-us/sysinfo... The system cannot find the file specified.
19 kernel32.dll 0x00000120 ReleaseMutex www.msdn.microsoft.com/library/en-us/dllproc/... Attempt to release mutex not owned b...
Details
Call Stack
Module Name Routine Name Source File Source Line
kernel32.dll ReleaseMutex 0
ResourceDemo.exe CResourceDemoApp::InitInstance ResourceDemo.cpp 69
ResourceDemo.exe WinMain appmodul.cpp 29
Event View Monitor Disassembler Editor Details Call Graph Call Tree

Like for the *Objects* category of profiling results (see above), the **Details** panel displays call stack information. In this panel, this is the stack of function calls that led to the error selected in the **Report** panel. If your application was compiled with debug information, you can view the source code of the routine selected in the call stack. For this purpose, just double-click the routine in the call stack.

Resource Profiler - Report Panel Columns

When you view results of the **Resource** profiler, the **Report** panel contents depend on the currently selected category in the **Explorer** panel (see description of the Resource profiler).

Classes Category

When this category is active, the Report panel shows information about resources types, whose instances exist at the moment the results are generated. This means, that if a resource instance was created and destroyed before the results were generated then it is not shown in the Report panel. The panel holds the following columns:

Columns (in alphabetical order)	**Description**
Image	The icon that indicates the category of resources of the given type.
Live Count	Number of resource instances that currently exist in memory.
Live Size	The size of currently existing resource instances (in bytes).

Module Name	Name of the module from which a Win32 API function that allocates resources was called.
Peak Created	Maximum number of concurrent resource instances reached during the run.
Peak Size	Maximum size of concurrent resource instances reached during the run (in bytes).
Class Name	Name of the resource type (*Registry*, *Menu*, *Handle*, *Bitmap*, etc.).
Total Created	Total number of resource instances that were created during the application run.
Total Size	Memory needed for all the resource instances that were created during the run.

Objects Category

When this category is active, the Report panel shows information about resource instances that exist at the moment that the results are generated. The panel holds the following columns:

Columns (in alphabetical order)	**Description**
#	The creation number of the given resource instance.
Address	Memory address at which the resource instance was allocated.
Get #	The ordinal number of the Get Results command that generated the current result set. For instance, if you pressed Get Results two times during the profiler run, you will get three result sets (the third will be generated after the application closes) with numbers 1, 2 and 3. The Get # value in all records of the first result set will hold 1; in the second result set this column will hold 2 and in the third result set the column will hold 3. The Get # column is used for comparison purposes. It lets you easily see which resource instances were created or deleted between two result generations.
Image	The icon that indicates the category of the given resource instance.
Module Name	Name of the module from which a Win32 API function that allocates resources was called.
Object Name	Name of the resource instance. It is formed as *Class Name* + *period* + *number*. For example, *Bitmap.3* means the third Bitmap resource instance that was created after the profiling started.
Class Name	Name of the resource type (*Registry*, *Menu*, *Handle*, *Bitmap*, etc.).
Size	Size of the resource instance in bytes.
Thread	Specifies the thread where the allocation routine of the resource instance was called.

Errors Category

When this category is active, the Report panel shows information about errors that occurred in resource management functions when the results are generated.For instance, if the `CreatePen` function returns NULL, which means that the function failed, the Report panel will hold a record that reports about this error. The panel holds the following columns:

Columns (in alphabetical order)	**Description**

#	The creation number of the given resource-related error.
Description	Text that describes the problem.
DLL Name	The name of the dynamic link library from which the API function is exported.
Error Code	The error code returned by the resource management function which caused the error.
Image	The icon that indicates the category of the given error (that is, whether it is an error or a warning).
Kind	The category of the resource management function that caused the error. The profiled function categories which the Resource profiler traces are specified by the profiler's *Resource categories to check* option.
Reference	The hyperlink to the MSDN topic concerning the API function. To open the topic, click the hyperlink.
Routine Name	Name of Win32 API's resource management function that caused the error.

Resource Profiler - Details Panel Columns

When you review the **Resource** profiler results, the Report panel displays information on resources that existed at the moment of results generation. The results that are shown in the **Report** and **Details** panels depend on the category selected in the **Explorer** panel.

If the *Objects* category is selected, the Report panel displays results for resource instances while the Details panel holds the stack of function calls that led to allocation of the resource instance selected in the Report panel (the API function that allocated the given resource instance is at the top of the call stack). If the *Errors* category is selected, the Report panel displays results for errors that occurred in API resource-related functions during the run while the Details panels contains the call stack for the error selected in the Report panel. In both instances, the call stack information is shown in the grid that has the following columns:

Columns (in alphabetical order)	**Description**
Module Name	Name of the module that holds the routine.
Routine Name	Name of the routine.
Source File	Name of the source file for the routine. The values for this column are read from the application's debug info.
Source Line	Source file's line number where the routine's implementation begins. The values for this column are read from the application's debug info.

Non-Existent Resources in the Report Panel

The **Resource** profiler reports whether the profiled application tried to release a non-existent resource. This can occur, for instance, if the application calls the `SysFreeString` function and passes it an invalid pointer (BSTR). However, sometimes AQtime may report that there was an attempt to free a non-allocated resource while this resource actually is allocated. Typically, this happens if you profile your executable(s) using the "Attach to Process" feature. If a resource was allocated before you attached to the desired process (that is, before you started profiling), AQtime will not "know" about it. If your application releases this resource before you finish profiling, AQtime will treat this as an attempt to free a non-allocated resource.

© 2010 AutomatedQA Corp.

Resource Profiler Options

The **Resource** profiler includes two groups of customizable options:

- One group contains options that affect the current result display. When you change these options, AQtime refreshes the data in its panels.
- Another group includes options that have effect on the profiler functioning. Changes in these options will only apply to future profiler runs.

To modify options that affect the result display, use items of the **Profiler** toolbar (by default, it is displayed at the top of the Report panel). If this toolbar is hidden, right-click somewhere in the toolbar area and select **Profiler** from the subsequent popup list. On the toolbar, the following items are available:

- **View project classes only** - Specifies whether profiling results are displayed only for the modules that are added to the Setup panel (enabled) or for all the modules used by the profiled application (disabled).

  This filter applies both to **Classes Data** and **Objects** categories.
- **Filter objects by stack** - Specifies whether to only display profiling results for objects created in the Setup modules (enabled), or for all modules used by the profiled application (disabled).

  This filter only applies to the **Objects** category.
- **Show all resources** - If this option is on, profiling results include all resource types being profiled. Otherwise, the results include only the resource types whose instances had been created before the results were generated.
- **Filter standard leaks** - If your application includes code that uses MFC, VCL or other libraries, some of the allocated Windows resources can not be released due to errors in the imported library code. If the Filter standard leaks option is enabled, AQtime excludes known resource leaks that were produced by third-party software from the profiling results. A list of known leaks is available at http://www.automatedqa.com/products/aqtime/leaks/.
- **Show routines with class names** - If it is enabled, the **Routine Name** column of the Details panels for the Resource profiler displays the name of the given routine along with the name of the class the routine belongs to. Otherwise, this column only displays the routine name.
- **File names with path** - If this option is enabled, the **Source File** and **Module Name** columns of the Report and Details panels for the Resource profiler hold the entire path to the given file. Otherwise, these columns hold the file name only.

To modify options that have effect on the way the profiler functions, do any of the following:

- *AQtime standalone:*
  - select **Options | Options** from AQtime's menu and then choose the **Profiles | Allocation | Resource Profiler** group from the ensuing Options dialog;
  - press **Configure Current Profiler** on the **Standard** toolbar when the **Resource** profiler is selected.
- *AQtime integrated into Visual Studio:*
  - select **Tools | Options** from Visual Studio's menu and then choose the **AQtime | Profilers | Allocation | Resource Profiler** group in the ensuing Options dialog.
- *AQtime integrated into Borland Developer Studio:*
  - select **Profile | Options** from Borland Developer Studio's menu and then choose the **Profilers | Allocation | Resource Profiler** group from the ensuing Options dialog.

Options include:

- **Collect stack information** - Specifies how the profiler should collect information on call stacks when allocating resource instances and tracing errors in resource management functions. The following values are available: *None*, *By routines* and *By lines*. Tracing the call stack can significantly slow down the profiled application. If you do not need to know the call stack, you can set this option to *None*. *By routines* means that the call stack entries will include information about routines only. If you want the call stack entries to include information on source line numbers as well, set the option to *By lines*. This will let you, for example, determine from which source line a function was called. Tracing source lines, however, requires time.
- **Thread model** - Specifies which thread model AQtime uses to trace the call stack for functions that allocate resources. For more information on supported values, see *Profiling Multiple Threads* and *Profiling COM Logical Threads*.
- **Resource categories to check** - Specifies the categories of resource management functions that should be traced during profiling. Available categories are:
  - *GDI and User Resources*
  - *GDI+ Resources*
  - *Kernel Resources*
  - *COM Resources*
  - *Registry Resources*
  - *Print Spooler Resources*

  To choose which of the categories to use, click the ellipisis button to the right of the option and check the desired categories in the subsequent dropdown list box.

Resource Profiler - List of Checked Functions

All the functions calls to which the **Resource** profiler traces are divided into five groups in accordance with the DLL a function is exported from:

- COM Resources (ole32.dll and oleaout32.dll)
- GDI and User Resources (gdi32.dll, user32.dll and shell32.dll)
- GDI+ Resources (gdiplus.dll)
- Kernel Resources (kernel32.dll, uxtheme.dll, wininet.dll and odbc32.dll)
- Print Spooler Resources (printspool.drv)
- Registry Resources (advapi32.dll and shlwapi.dll)

Using the *Resource categories to check* option of the Resource profiler, you can specify whether the profiler should track all the functions of this or that category.

"COM Resources" Function Category (ole32.dll and oleaout32.dll)

CoInitialize

CoInitializeEx

CoTaskMemAlloc

CoTaskMemFree

CoTaskMemRealloc

CoUninitialize

OleInitialize
OleUninitialize
SafeArrayAllocDescriptor
SafeArrayAllocDescriptorEx
SafeArrayCopy
SafeArrayCreate
SafeArrayCreateEx
SafeArrayCreateVector
SafeArrayCreateVectorEx
SafeArrayDestroy
SafeArrayDestroyData
SafeArrayDestroyDescriptor
SafeArrayGetElement
StringFromCLSID
StringFromIID
SysAllocString
SysAllocStringByteLen
SysAllocStringLen
SysFreeString
SysReAllocString
SysReAllocStringLen
VariantClear
VariantCopy

VariantCopyInd **"GDI and User Resources" Function Category (gdi32.dll, user32.dll and shell32.dll)**

CallWindowProcA
CallWindowProcW
CloseEnhMetaFile
CloseMetaFile
CloseWindowStation
CopyCursor
CopyEnhMetaFileA
CopyEnhMetaFileW
CopyIcon
CopyImage
CopyMetaFileA
CopyMetaFileW
CreateAcceleratorTableA

CreateAcceleratorTableW
CreateBitmap
CreateBitmapIndirect
CreateBrushIndirect
CreateColorSpaceA
CreateColorSpaceW
CreateCompatibleBitmap
CreateCompatibleDC
CreateCursor
CreateDCA
CreateDCW
CreateDialogIndirectParamA
CreateDialogIndirectParamW
CreateDialogParamA
CreateDialogParamW
CreateDIBitmap
CreateDIBPatternBrush
CreateDIBPatternBrushPt
CreateDIBSection
CreateDiscardableBitmap
CreateEllipticRgn
CreateEllipticRgnIndirect
CreateEnhMetaFileA
CreateEnhMetaFileW
CreateFontA
CreateFontW
CreateFontIndirectA
CreateFontIndirectW
CreateFontIndirectExA
CreateFontIndirectExW
CreateHalftonePalette
CreateHatchBrush
CreateICA
CreateICW
CreateIcon
CreateIconFromResourceEx
CreateIconIndirect
CreateMDIWindowA
CreateMDIWindowW

© 2010 AutomatedQA Corp.

CreateMenu
CreateMetaFileA
CreateMetaFileW
CreatePalette
CreatePatternBrush
CreatePen
CreatePenIndirect
CreatePolygonRgn
CreatePolyPolygonRgn
CreatePopupMenu
CreateRectRgn
CreateRectRgnIndirect
CreateRoundRectRgn
CreateSolidBrush
CreateWindowA
CreateWindowW
CreateWindowExA
CreateWindowExW
CreateWindowStationA
CreateWindowStationW
DefFrameProcA
DefFrameProcW
DefMDIChildProcA
DefMDIChildProcW
DefWindowProcA
DefWindowProcW
DeleteColorSpace
DeleteDC
DeleteEnhMetaFile
DeleteMetaFile
DeleteObject
DestroyAcceleratorTable
DestroyCursor
DestroyIcon
DestroyMenu
DestroyWindow
DuplicateIcon
ExtCreatePen
ExtCreateRegion

ExtractAssociatedIconA
ExtractAssociatedIconW
ExtractAssociatedIconExA
ExtractAssociatedIconExW
ExtractIconA
ExtractIconW
ExtractIconExA
ExtractIconExW
GdipGetDC
GdipReleaseDC
GetClassInfoA
GetClassInfoW
GetClassInfoEx A
GetClassInfoExW
GetDC
GetDCEx
GetEnhMetaFileA
GetEnhMetaFileW
GetIconInfo
GetMetaFileA
GetMetaFileW
GetProcAddress
GetWindowDC
InsertMenuA
InsertMenuW
InsertMenuItemA
InsertMenuItemW
LoadBitmapA
LoadBitmapW
LoadCursorFromFileA
LoadCursorFromFileW
LoadImageA
LoadImageW
LoadKeyboardLayoutA
LoadKeyboardLayoutW
LoadMenuA
LoadMenuW
LoadMenuIndirectA
LoadMenuIndirectW

© 2010 AutomatedQA Corp.

OpenWindowStationA
OpenWindowStationW
RegisterClassA
RegisterClassW
RegisterClassExA
RegisterClassExW
ReleaseDC
ReleaseStgMedium
SetClipboardData
SetEnhMetaFileBits
SetMetaFileBitsEx
SetWindowRgn
SetWinMetaFileBits
SHFileOperation
SHFileOperationA
SHFileOperationW
SHFreeNameMappings
SHGetFileInfo
SHGetFileInfoA
SHGetFileInfoW
UnloadKeyboardLayout

"GDI+ Resources" Function Category (gdiplus.dll)

GdipCloneCustomLineCap
GdipCloneFont
GdipCloneFontFamily
GdipCloneImage
GdipCloneMatrix
GdipClonePath
GdipClonePen
GdipCloneRegion
GdipCloneStringFormat
GdipCreateCustomLineCap
GdipCreateFont
GdipCreateFontFamilyFromName
GdipCreateFontFromDC
GdipCreateFontFromLogfontA
GdipCreateFontFromLogfontW
GdipCreateFromHDC

GdipCreateFromHDC2
GdipCreateFromHWND
GdipCreateFromHWNDICM
GdipCreateMatrix
GdipCreateMatrix2
GdipCreateMatrix3
GdipCreateMatrix3I
GdipCreatePath
GdipCreatePath2
GdipCreatePath2I
GdipCreatePen1
GdipCreatePen2
GdipCreateRegion
GdipCreateRegionHrgn
GdipCreateRegionPath
GdipCreateRegionRect
GdipCreateRegionRectI
GdipCreateRegionRgnData
GdipCreateStringFormat
GdipDeleteCustomLineCap
GdipDeleteFont
GdipDeleteFontFamily
GdipDeleteGraphics
GdipDeleteMatrix
GdipDeletePath
GdipDeletePen
GdipDeleteRegion
GdipDeleteStringFormat
GdipDisposeImage
GdipLoadImageFromFile
GdipLoadImageFromFileICM
GdipLoadImageFromStream
GdipLoadImageFromStreamICM

"Kernel Resources" Function Category (kernel32.dll, uxtheme.dll, wininet.dll and odbc32.dll)

_lclose
_lcreat
_lopen

BeginUpdateResourceA
BeginUpdateResourceW
CloseEventLog
CloseHandle
CloseThemeData
CreateConsoleScreenBuffer
CreateEventA
CreateEventW
CreateFiber
CreateFiberEx
CreateFileA
CreateFileW
CreateFileMappingA
CreateFileMappingW
CreateMailslotA
CreateMailslotW
CreateMutexA
CreateMutexW
CreateNamedPipeA
CreateNamedPipeW
CreatePipe
CreateProcessA
CreateProcessW
CreateProcessAsUserA
CreateProcessAsUserW
CreateProcessWithLogonW
CreateProcessWithTokenW
CreateRemoteThread
CreateSemaphoreA
CreateSemaphoreW
CreateThread
CreateTransaction
DeleteCriticalSection
DeleteFiber
DeregisterEventSource
DuplicateHandle
EndUpdateResourceA
EndUpdateResourceW
FindClose

FindCloseChangeNotification
FindFirstChangeNotificationA
FindFirstChangeNotificationW
FindFirstFileA
FindFirstFileW
FindFirstFileExA
FindFirstFileExW
FtpFindFirstFileA
FtpFindFirstFileW
FtpOpenFileA
FtpOpenFileW
GetThemeSysColorBrush
GopherFindFirstFileA
GopherFindFirstFileW
GopherOpenFileA
GopherOpenFileW
HttpOpenRequestA
HttpOpenRequestW
InitializeCriticalSection
InternetConnectA
InternetConnectW
InternetOpenA
InternetOpenW
MapViewOfFile
MapViewOfFileEx
OpenBackupEventLogA
OpenBackupEventLogW
OpenEventA
OpenEventW
OpenEventLog A
OpenEventLogW
OpenFile
OpenFileMappingA
OpenFileMappingW
OpenMutexA
OpenMutexW
OpenProcess
OpenSemaphoreA
OpenSemaphoreW

© 2010 AutomatedQA Corp.

OpenThemeData
OpenTransaction
RegisterEventSourceA
RegisterEventSourceW
ReleaseMutex
ReleaseSemaphore
RetrieveUrlCacheEntryStreamA
RetrieveUrlCacheEntryStreamW
SQLAllocConnect
SQLAllocEnv
SQLAllocHandle
SQLAllocStmt
SQLFreeConnect
SQLFreeEnv
SQLFreeHandle
SQLFreeStmt
TerminateThread
TlsAlloc
TlsFree
UnlockUrlCacheEntryStream
UnmapViewOfFile

"Print Spooler Resources" Function Category (printspool.drv)

AddPrinterA
AddPrinterW
ClosePrinter
OpenPrinterA
OpenPrinterW

"Registry Resources" Function Category (advapi32.dll and shlwapi.dll)

RegCloseKey
RegConnectRegistryA
RegConnectRegistryW
RegCreateKeyA
RegCreateKeyW
RegCreateKeyExA
RegCreateKeyExW
RegCreateKeyTransacted
RegOpenKeyA

RegOpenKeyW
RegOpenKeyExA
RegOpenKeyExW
RegOpenKeyTransacted
SHRegCloseUSKey
SHRegCreateUSKeyA
SHRegCreateUSKeyW
SHRegOpenUSKeyA
SHRegOpenUSKeyW

Sequence Diagram Link Profiler

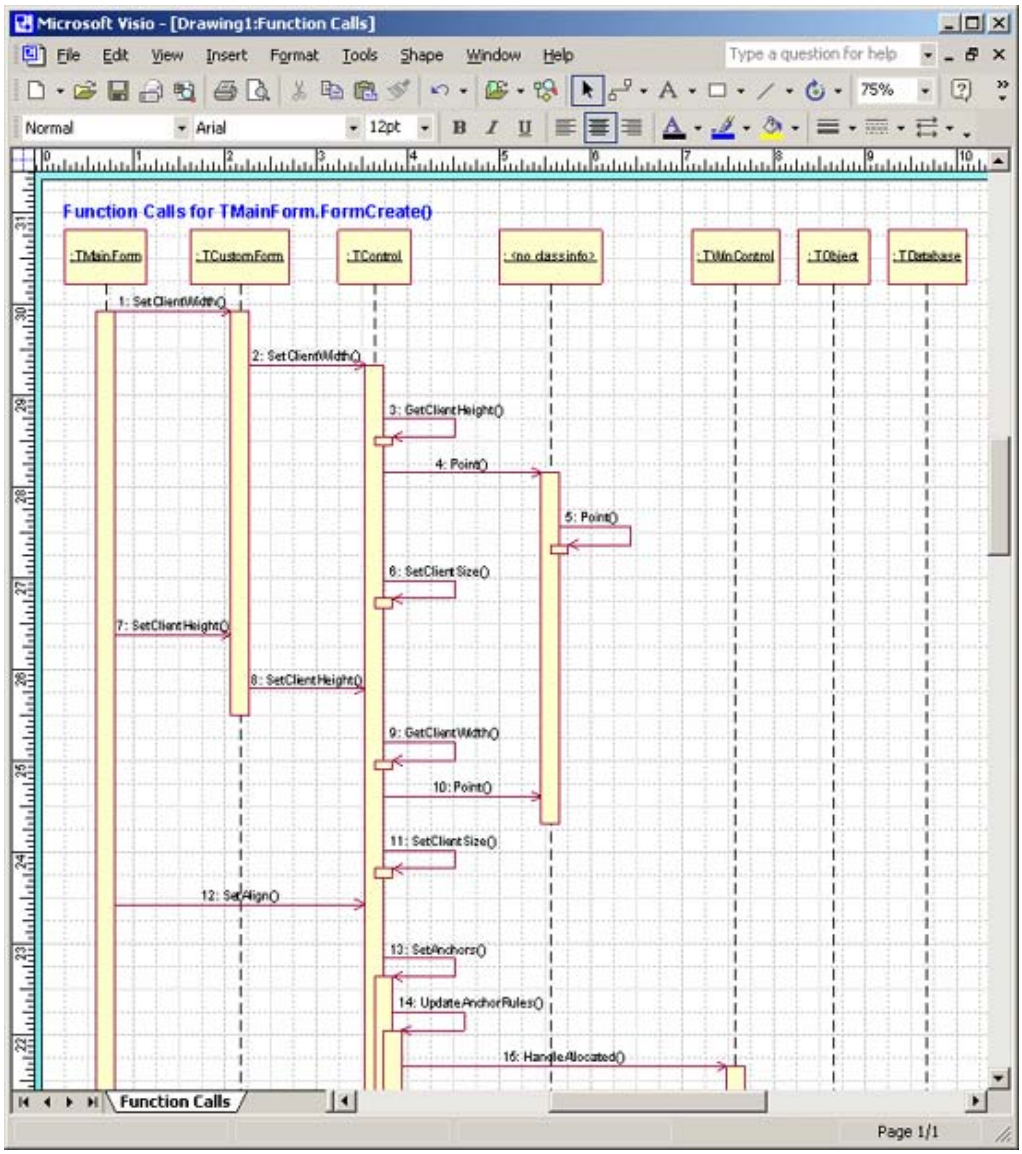
Sequence Diagram Link Profiler - Overview

The **Sequence Diagram Link** profiler analyzes the sequence of function calls in your application and then builds a UML-style diagram of function calls in Microsoft Word or Microsoft Visio. It is a convenient tool to trace links between methods and functions without running the application. Note that these are potential links between routines, since the profiler statically analyzes your application and it cannot predict whether conditional calls will be performed.

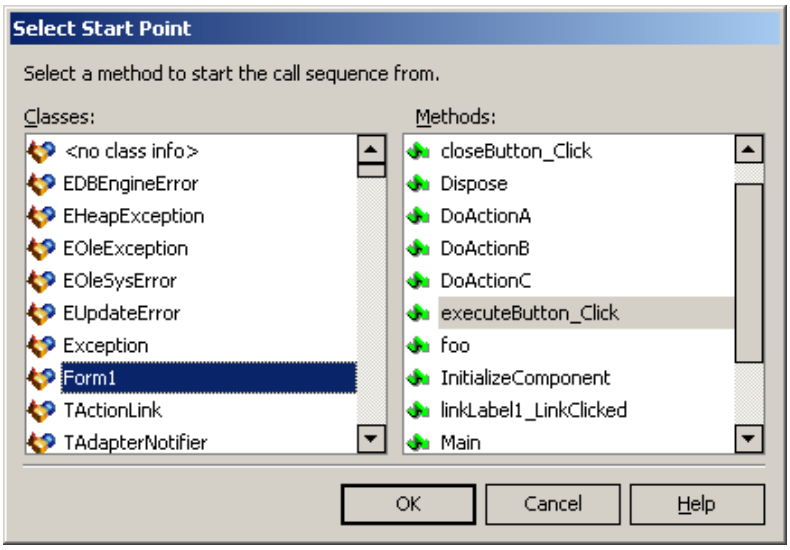
Currently, the Sequence Diagram Link profiler supports the following versions of Microsoft Word and Microsoft Visio:

- Microsoft Word 97 or higher
- Microsoft Visio 2000 or higher

The program in which the diagram will be created is specified by the *Target application* option of the profiler. For example, the following diagram was created in Microsoft Visio:



The profiler algorithm is based on backtracking. After the profiler has been executed, it displays the **Select Start Point** dialog:



Here you can select the routine, which will be the starting point of analysis. The Sequence Diagram Link profiler parses the binary code of this routine and determines the functions called from it. Then it parses the first function found and, if it finds calls to child function, it continues to the first function it finds. The analysis is performed recursively until the profiler finds a function, which has no child function calls. When all functions at a given level have been analyzed, the profiler moves up one level and continues with the next function.

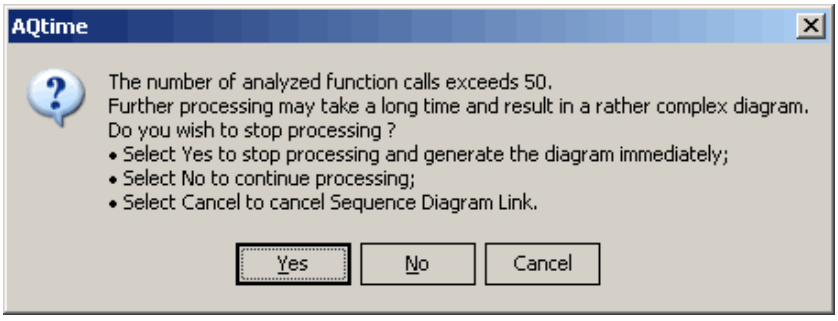
The Sequence Diagram Link profiler supports profiling areas of the routine level (the class- and line-level areas as well as the **Profile Entire .NET Code** settings are ignored).

If a routine was not included in profiling, the profiler does not parse it and does not parse calls to its child functions. Note however, that the Sequence Diagram Link analyzes only routines that are located in the module to which the start-point routine belongs. It will ignore all other routines, even if they are specified in the "Including" profiling areas.

The Sequence Diagram Link ignores calls to abstract and interface methods. The reason for this is that debug info does not contain information on these methods, so the profiler is unable to analyze them.

Calls to properties are displayed as calls to property's *read* or *write* methods (`get_PropertyName` and `set_PropertyName`). If a property does not have a *read* (*write*) method, calls to this property are ignored.

The profiler can work until it parses all functions in the module. However, to avoid the creation overly complex diagrams, AQtime lets you stop the analysis when the number of parsed routines exceeds the value specified by the *Warning level* profiler option:


AQtime
The number of analyzed function calls exceeds 50.
Further processing may take a long time and result in a rather complex diagram.
Do you wish to stop processing ?
• Select Yes to stop processing and generate the diagram immediately;
• Select No to continue processing;
• Select Cancel to cancel Sequence Diagram Link.
Yes
No
Cancel

Sequence Diagram Link Profiler Options

The **Sequence Diagram Link** profiler includes a number of options to customize. To modify them, do any of the following:

- *AQtime standalone:*
  - select **Options | Options** from AQtime's main menu (this will call the Options dialog), and then choose **Profilers | Static Analysis | Sequence Diagram Link** from the tree view on the left of the dialog;
  - press **Configure Current Profiler** on the **Standard** toolbar when the **Sequence Diagram Link** profiler is selected.
- *AQtime integrated into Visual Studio:*
  - select **Tools | Options** from Visual Studio's main menu (this will call the Options dialog) and then select **AQtime | Profilers | Static Analysis | Sequence Diagram Link** from the tree view on the left of the dialog.
- *AQtime integrated into Borland Developer Studio:*
  - select **Profile | Options** from Borland Developer Studio's main menu (this will call the Options dialog), and then choose **Profilers | Static Analysis | Sequence Diagram Link** from the tree view on the left of the dialog.

In the dialog, the following options are available:

- **Output diagram options**
  - **Draw message numbers** - If this option is enabled, AQtime draws numbers next to function names in the output diagram. Otherwise, the numbers are not drawn.
- **Warning level** - The number of analyzed methods, after which AQtime displays a dialog box asking if you want to continue profiling. This option helps to avoid the creation of overly complex diagrams and saves its results. Possible values are between 0 and 1,000,000. Default: 50. 0 = never ask.
- **Target application** - The application, used to create the diagram of function calls (*Microsoft Word* or *Microsoft Visio*).

Static Analysis Profiler

Static Analysis Profiler - Overview

This topic provides the **Static Analysis** profiler overview and describes the profiler results. The complete profiler description includes the following topics:

- *Overview (this topic)*
- *Static Analysis Profiler - Analyzing Results*
- *Static Analysis Profiler - Report Panel Columns*
- *Static Analysis Profiler - Columns of the Details and Call Tree Panels*
- *Static Analysis Profiler Options*

Overview

The **Static Analysis** profiler (as the word “static” indicates) does not launch the tested application, but analyzes the debugging information (native applications) or metadata (.NET-connected applications) that is included in the executable(s) to find information as:

- the size of the routines in bytes,
- their length in source code lines,
- the routine addresses in memory,
- the structure of method calls as written in the source code,
- the binary code generated for the routine,
- the number of binary instructions in the routine,
- the number of loop instructions in the routine,
- the number of exception handling frames in the routine,
- the number of conditional operators in the routine,
- the number of floating point operators in the routine,
- and so forth...

In addition, the Static Analysis finds all of the potential interlinks of your application’s classes through their methods, that is, the links in all possible code branches. Therefore, the Static Analysis Profiler centers on both methods (routines) and class interlinks.

The Static Analysis profiler supports both 32-bit and 64-bit applications.

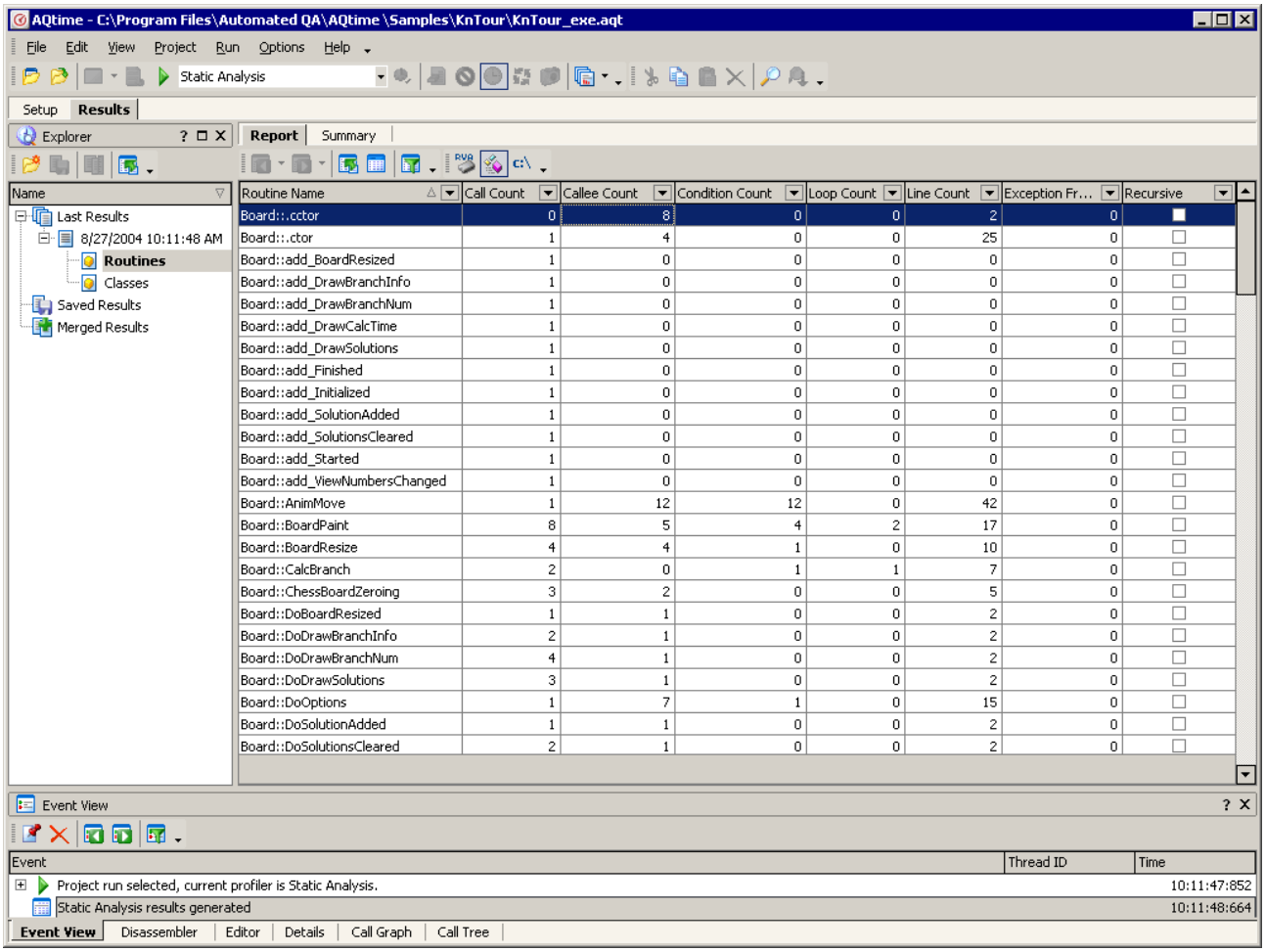
When you start profiling with the Static Analysis profiler, the application does not execute; the profiler simply checks the executable(s) included in the current AQtime project. Area and trigger settings are ignored. Some questions that can be answered by this speedy analysis are:

- What code is used by an application? If the application includes a massive module only to use one or two functions from it, you might choose to extract them from the module, or to re-implement them so as to save on application size and dependencies.
- Which routine is located by a certain address? For instance, if the application raises an exception, you can launch Static Analysis and determine from the exception address reported which routine caused it.
- What binary code was produced by the compiler for a routine? This can tell you for instance if array or string parameters are being passed by copying the data to the stack, or only a pointer.
- What functions and procedures are called by a routine? This tells you what methods are called or (more exactly) can be called from other methods.
- Is a routine overburdened with too many loops, conditional jumps, exception handling frames and other code structures that may impede the routine’s performance? Such routines are potential candidates to be rewritten.
- What classes exist in the application, what methods they have, which methods of other classes call methods of the given class or are called by this class's methods (in source code, independent of whether the call is ever executed).

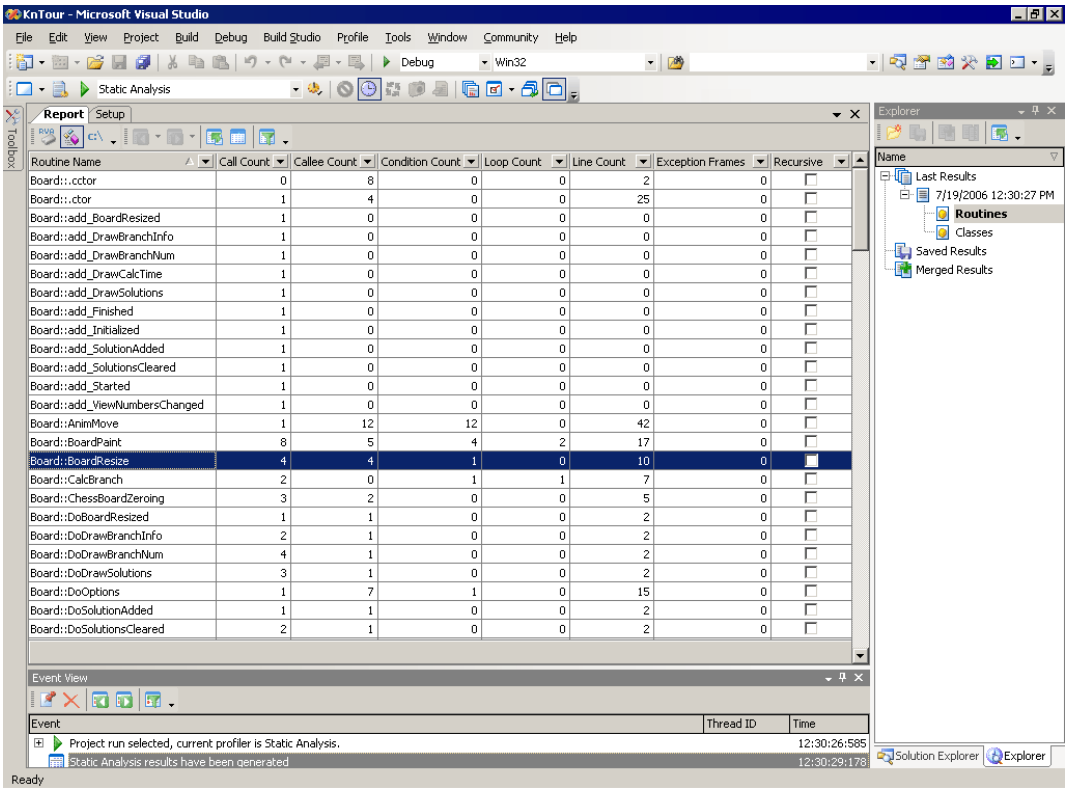
The **Report** panel is the “main” results display. The Static Analysis profiler results are divided into two categories: **Routines** and **Classes**. These categories are displayed as subnodes of the result set node in the **Explorer** panel. The contents of the other panels (Report, Details, Call Graph, etc.) depend on the currently selected category.

- **Routines**. When this category is selected, the Report panel displays information about routines that can be potentially called. Each row shows profiling results for every single routine: the call

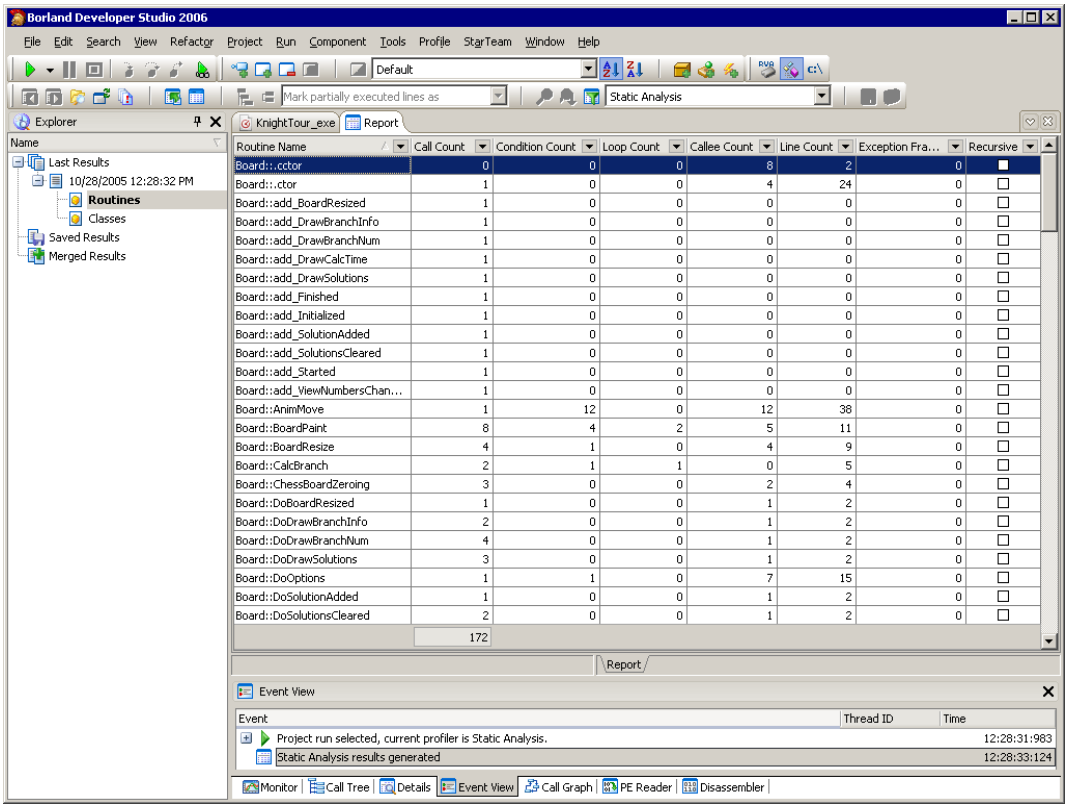
count, address, size, etc. (For more detailed information, review *Static Analysis Profiler - Report Panel Columns*). This gives you an entire picture of routine calls in your application. (Note that you can also view the summary results in the **Summary** panel. It reports about routines that are called most often, largest routines (in source code lines and in bytes), routines with maximum number of binary instructions, etc.)


AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 10:11:48 AM
Routines
Classes
Saved Results
Merged Results
Routine Name | Call Count | Callee Count | Condition Count | Loop Count | Line Count | Exception Fr... | Recursive
Board::.cctor 0 8 0 0 2 0
Board::.ctor 1 4 0 0 25 0
Board::add_BoardResized 1 0 0 0 0 0
Board::add_DrawBranchInfo 1 0 0 0 0 0
Board::add_DrawBranchNum 1 0 0 0 0 0
Board::add_DrawCalcTime 1 0 0 0 0 0
Board::add_DrawSolutions 1 0 0 0 0 0
Board::add_Finished 1 0 0 0 0 0
Board::add_Initialized 1 0 0 0 0 0
Board::add_SolutionAdded 1 0 0 0 0 0
Board::add_SolutionsCleared 1 0 0 0 0 0
Board::add_Started 1 0 0 0 0 0
Board::add_ViewNumbersChanged 1 0 0 0 0 0
Board::AnimMove 1 12 12 0 42 0
Board::BoardPaint 8 5 4 2 17 0
Board::BoardResize 4 4 1 0 10 0
Board::CalcBranch 2 0 1 1 7 0
Board::ChessBoardZeroing 3 2 0 0 5 0
Board::DoBoardResized 1 1 0 0 2 0
Board::DoDrawBranchInfo 2 1 0 0 2 0
Board::DoDrawBranchNum 4 1 0 0 2 0
Board::DoDrawSolutions 3 1 0 0 2 0
Board::DoOptions 1 7 1 0 15 0
Board::DoSolutionAdded 1 1 0 0 2 0
Board::DoSolutionsCleared 2 1 0 0 2 0
Event View
Event | Thread ID | Time
Project run selected, current profiler is Static Analysis. 10:11:47:852
Static Analysis results generated 10:11:48:664
Event View Disassembler Editor Details Call Graph Call Tree

AQtime standalone

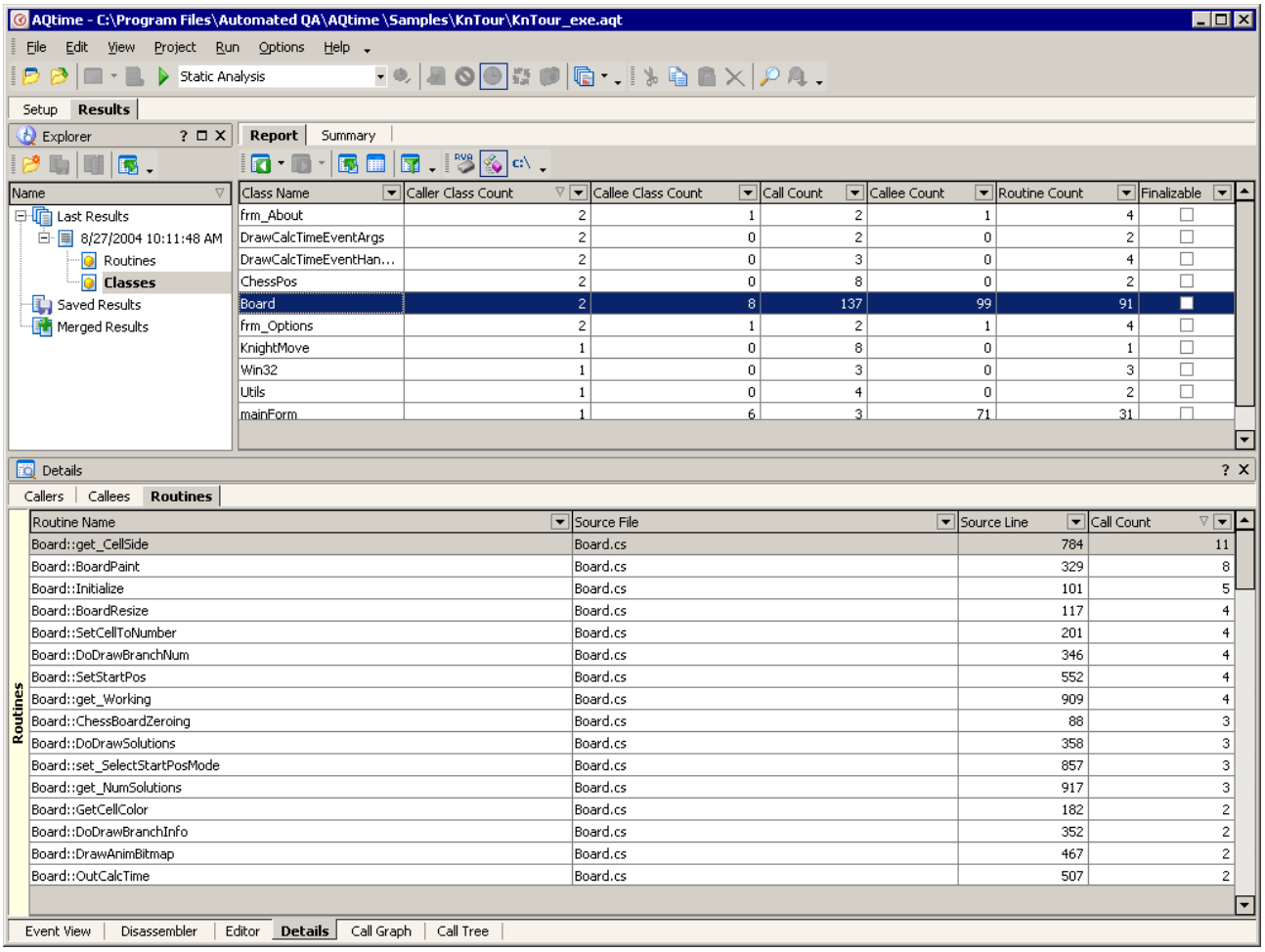


AQtime integrated into Visual Studio



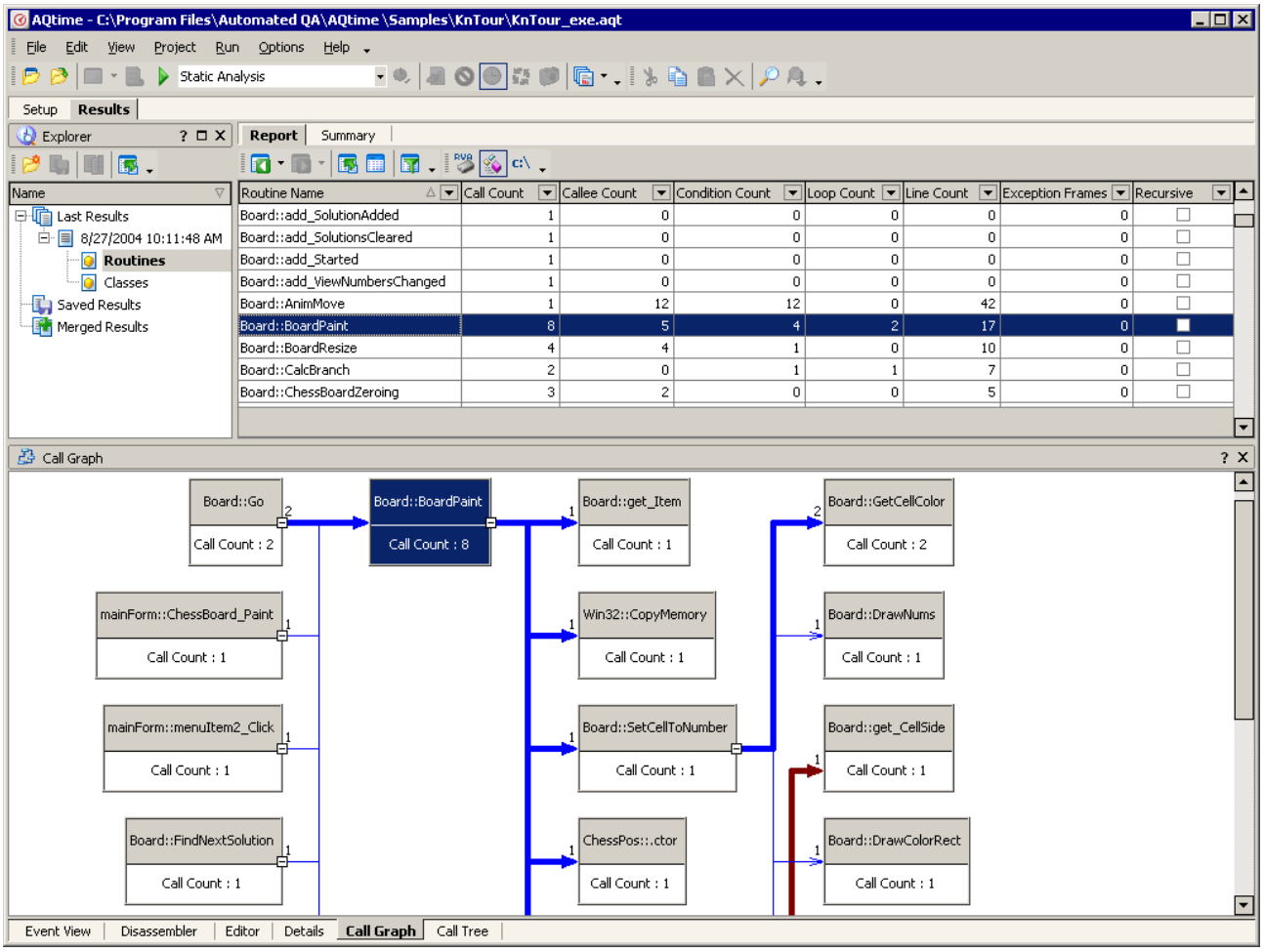
AQtime integrated into Borland Developer Studio

- **Classes**. When this category is selected, the Report panel displays information about class interlinks (through potential calls between class methods) in your application. Every row in the panel holds results for a single class: the number of methods in the class, the number of classes whose methods call methods of the given class, the number of classes whose methods are called by methods of the given class, etc. For more information about the Report panel columns, see *Static Analysis Profiler - Report Panel Columns*.

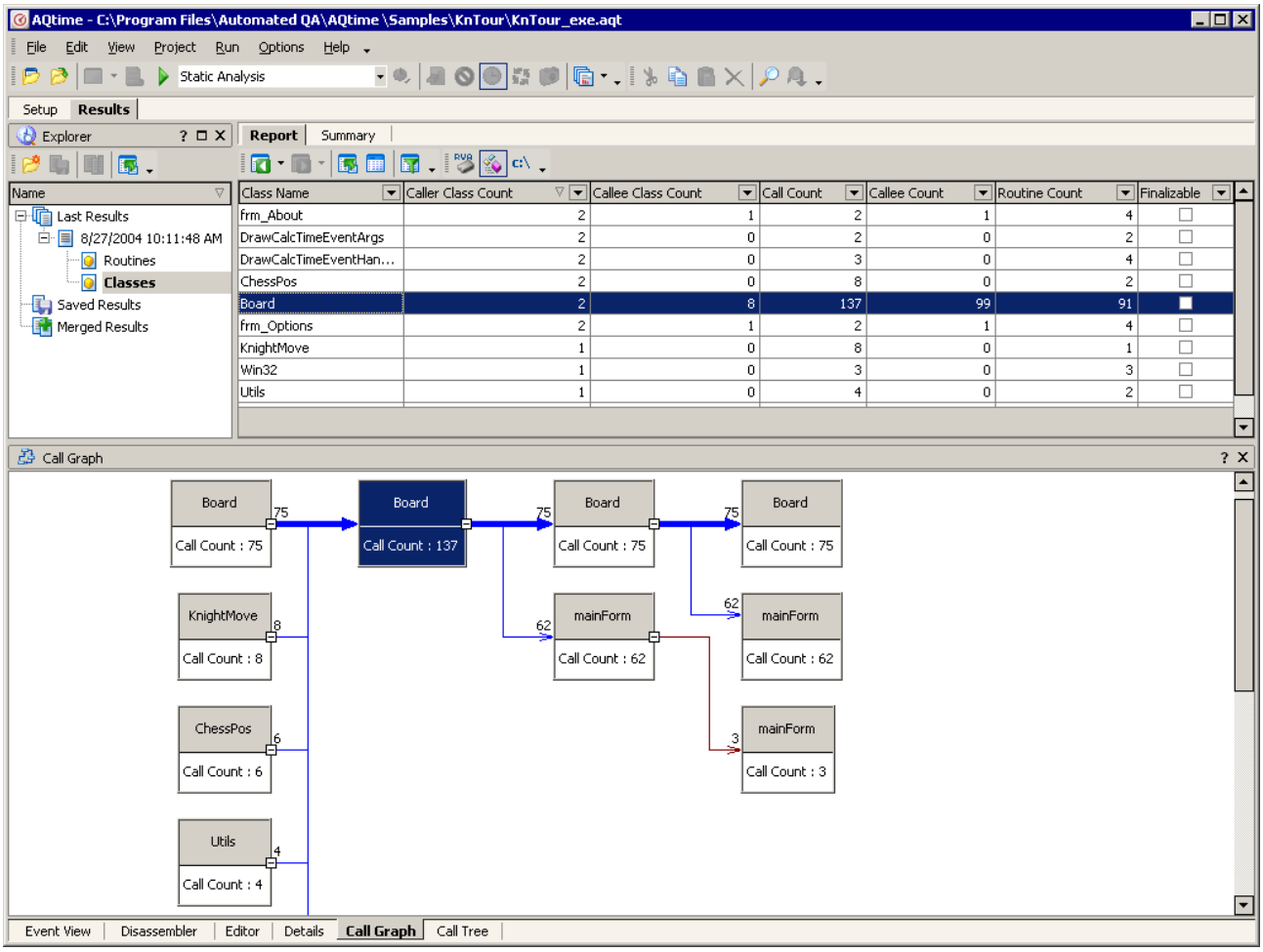

AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 10:11:48 AM
Routines
Classes
Saved Results
Merged Results
Class Name | Caller Class Count | Callee Class Count | Call Count | Callee Count | Routine Count | Finalizable
frm_About 2 1 2 1 4
DrawCalcTimeEventArgs 2 0 2 0 2
DrawCalcTimeEventHan... 2 0 3 0 4
ChessPos 2 0 8 0 2
Board 2 8 137 99 91
frm_Options 2 1 2 1 4
KnightMove 1 0 8 0 1
Win32 1 0 3 0 3
Utils 1 0 4 0 2
mainForm 1 6 3 71 31
Details
Callers Callees Routines
Routines
Routine Name | Source File | Source Line | Call Count
Board::get_CellSide Board.cs 784 11
Board::BoardPaint Board.cs 329 8
Board::Initialize Board.cs 101 5
Board::BoardResize Board.cs 117 4
Board::SetCellToNumber Board.cs 201 4
Board::DoDrawBranchNum Board.cs 346 4
Board::SetStartPos Board.cs 552 4
Board::get_Working Board.cs 909 4
Board::ChessBoardZeroing Board.cs 88 3
Board::DoDrawSolutions Board.cs 358 3
Board::set_SelectStartPosMode Board.cs 857 3
Board::get_NumSolutions Board.cs 917 3
Board::GetCellColor Board.cs 182 2
Board::DoDrawBranchInfo Board.cs 352 2
Board::DrawAnimBitmap Board.cs 467 2
Board::OutCalcTime Board.cs 507 2
Event View Disassembler Editor Details Call Graph Call Tree

You can arrange the Report panel the same way you organize the other AQtime panels.

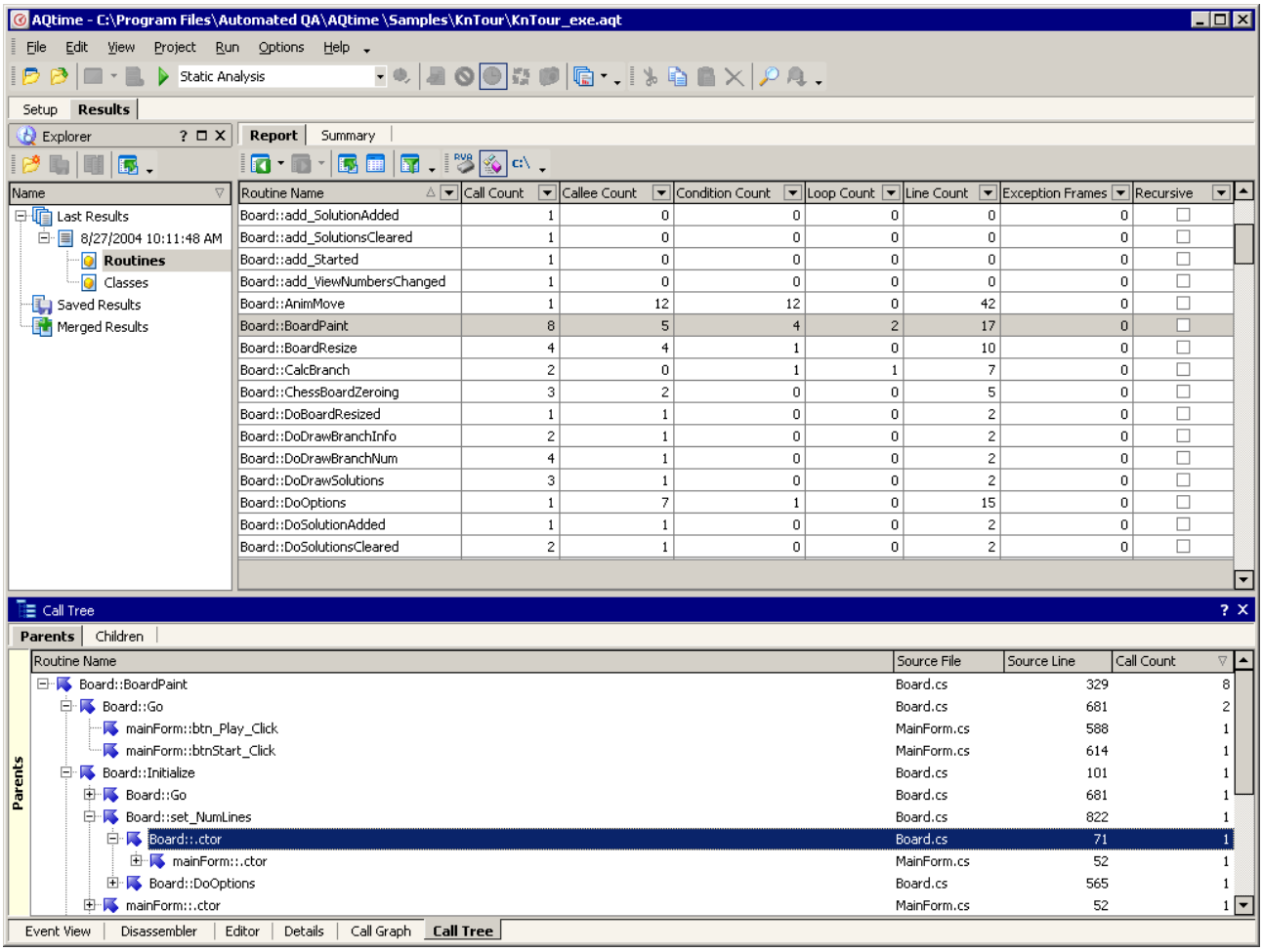
For the Routines category, Static Analysis finds how routine calls are coded in source – what is called (child) from where (parent). The **Call Graph** panel displays this hierarchy in a convenient, browsable format. Click a method line in Report and the method's parent callers and child callees will be shown in the Call Graph with call counts (the number of points where calls occur in the source).


AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 10:11:48 AM
Routines
Classes
Saved Results
Merged Results
Routine Name | Call Count | Callee Count | Condition Count | Loop Count | Line Count | Exception Frames | Recursive
Board::add_SolutionAdded 1 0 0 0 0 0
Board::add_SolutionsCleared 1 0 0 0 0 0
Board::add_Started 1 0 0 0 0 0
Board::add_ViewNumbersChanged 1 0 0 0 0 0
Board::AnimMove 1 12 12 0 42 0
Board::BoardPaint 8 5 4 2 17 0
Board::BoardResize 4 4 1 0 10 0
Board::CalcBranch 2 0 1 1 7 0
Board::ChessBoardZeroing 3 2 0 0 5 0
Call Graph
Board::Go Call Count : 2
Board::BoardPaint Call Count : 8
Board::get_Item Call Count : 1
Board::GetCellColor Call Count : 2
mainForm::ChessBoard_Paint Call Count : 1
Win32::CopyMemory Call Count : 1
Board::DrawNums Call Count : 1
mainForm::menuItem2_Click Call Count : 1
Board::SetCellToNumber Call Count : 1
Board::get_CellSide Call Count : 1
Board::FindNextSolution Call Count : 1
ChessPos:::.ctor Call Count : 1
Board::DrawColorRect Call Count : 1
Event View Disassembler Editor Details Call Graph Call Tree

For the Classes category, the Call Graph displays the hierarchy of method interlinks between classes. Click a class line in the Report pane, and the classes whose methods call the given class's methods or are called by these methods will be shown in the Call Graph.

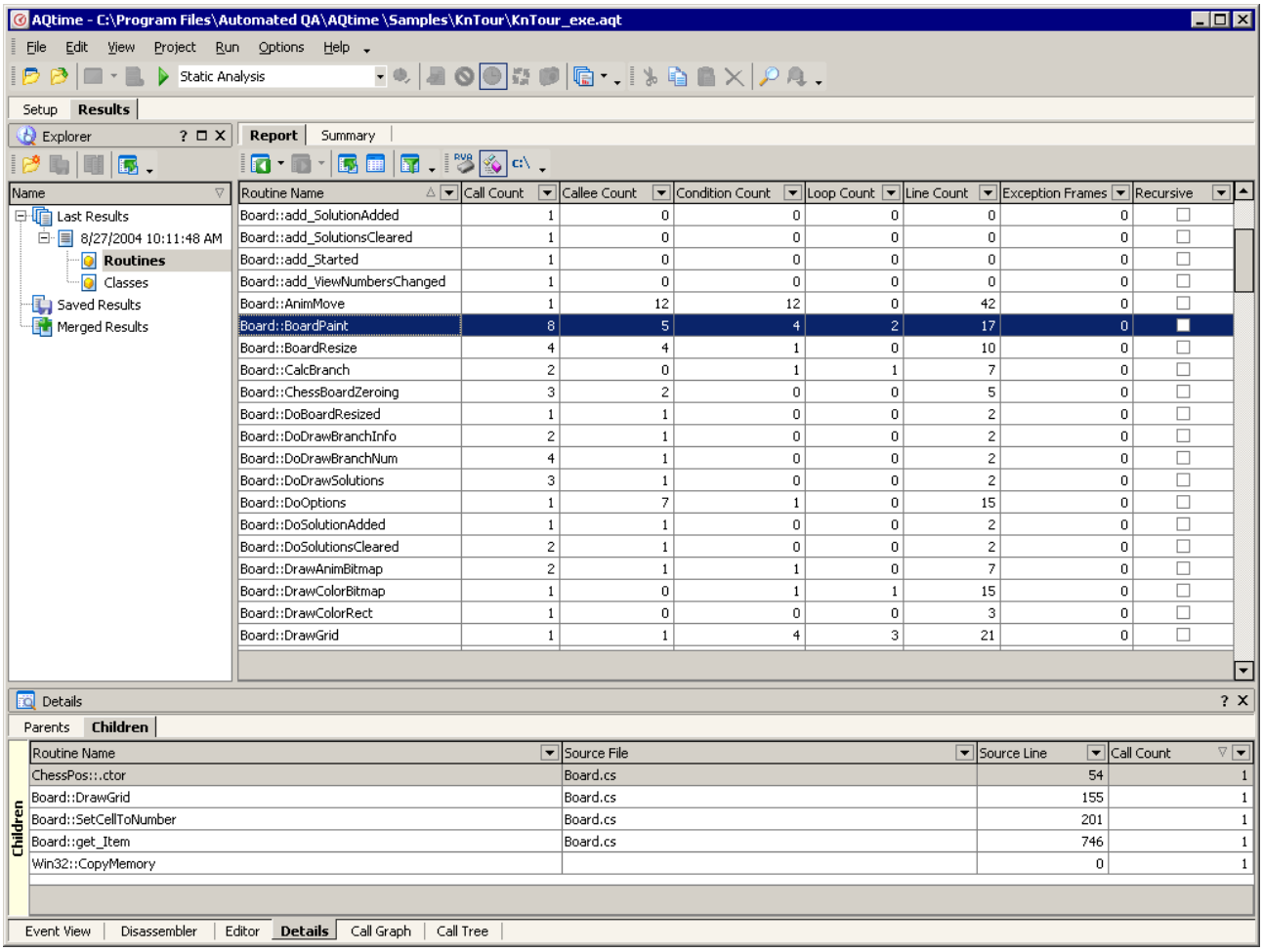

AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 10:11:48 AM
Routines
Classes
Saved Results
Merged Results
Class Name | Caller Class Count | Callee Class Count | Call Count | Callee Count | Routine Count | Finalizable
frm_About 2 1 2 1 4
DrawCalcTimeEventArgs 2 0 2 0 2
DrawCalcTimeEventHan... 2 0 3 0 4
ChessPos 2 0 8 0 2
Board 2 8 137 99 91
frm_Options 2 1 2 1 4
KnightMove 1 0 8 0 1
Win32 1 0 3 0 3
Utils 1 0 4 0 2
Call Graph
Board Call Count : 75
Board Call Count : 137
Board Call Count : 75
Board Call Count : 75
KnightMove Call Count : 8
mainForm Call Count : 62
mainForm Call Count : 62
ChessPos Call Count : 6
mainForm Call Count : 3
Utils Call Count : 4
75
75
75
8
62
62
6
3
4
Event View Disassembler Editor Details Call Graph Call Tree

An alternative view for the same information is given in the **Call Tree** panel. This panel provides a tree-view of the hierarchy of routine calls or class interlinks (see *Static Analysis Profiler - Columns of the Details and Call Tree Panels*). For instance, the following picture illustrates how the routine call hierarchy looks in the Call Tree panel.

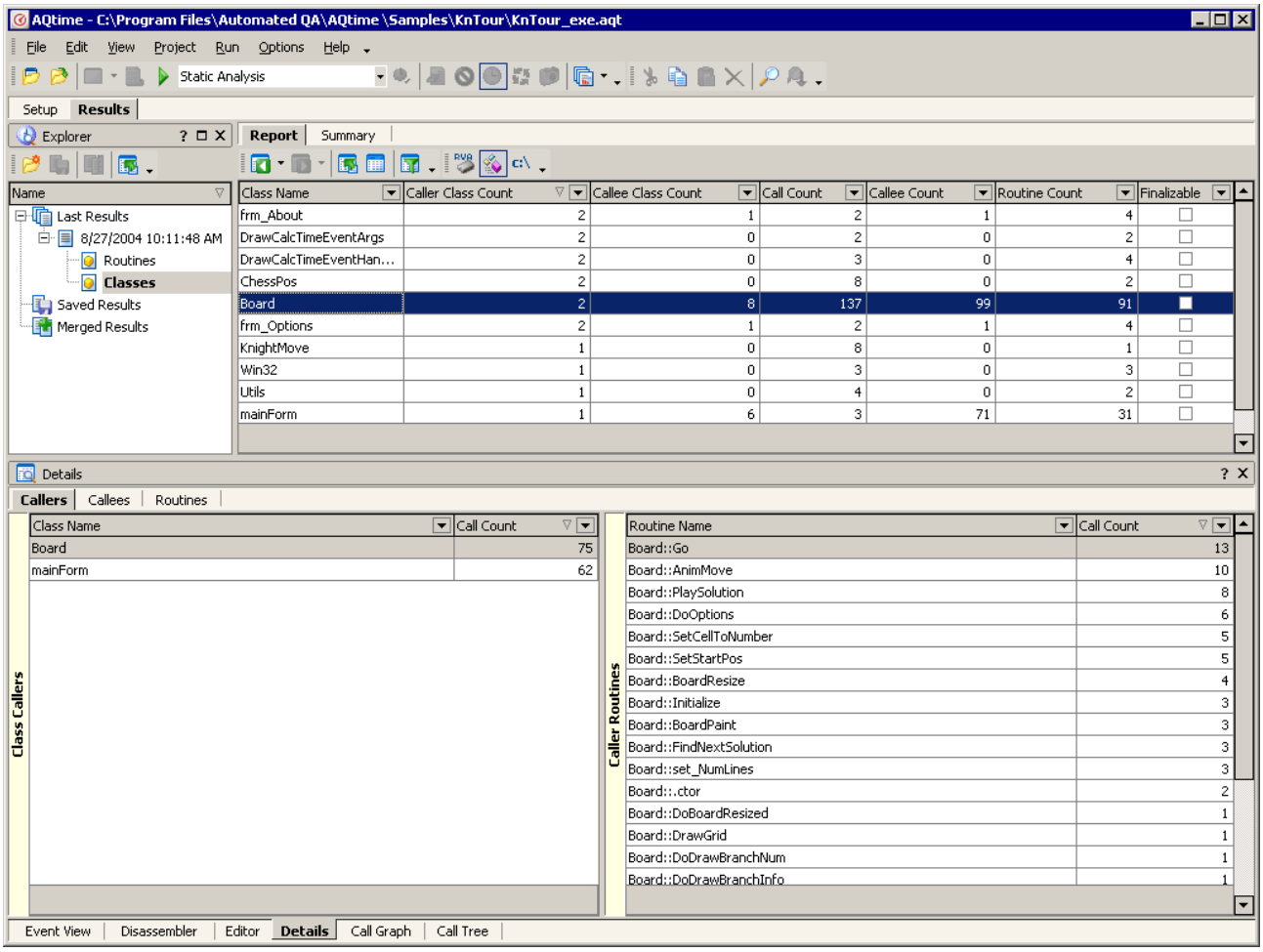

AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 10:11:48 AM
Routines
Classes
Saved Results
Merged Results
Routine Name Call Count Callee Count Condition Count Loop Count Line Count Exception Frames Recursive
Board::add_SolutionAdded 1 0 0 0 0 0
Board::add_SolutionsCleared 1 0 0 0 0 0
Board::add_Started 1 0 0 0 0 0
Board::add_ViewNumbersChanged 1 0 0 0 0 0
Board::AnimMove 1 12 12 0 42 0
Board::BoardPaint 8 5 4 2 17 0
Board::BoardResize 4 4 1 0 10 0
Board::CalcBranch 2 0 1 1 7 0
Board::ChessBoardZeroing 3 2 0 0 5 0
Board::DoBoardResized 1 1 0 0 2 0
Board::DoDrawBranchInfo 2 1 0 0 2 0
Board::DoDrawBranchNum 4 1 0 0 2 0
Board::DoDrawSolutions 3 1 0 0 2 0
Board::DoOptions 1 7 1 0 15 0
Board::DoSolutionAdded 1 1 0 0 2 0
Board::DoSolutionsCleared 2 1 0 0 2 0
Call Tree
Parents Children
Parents
Routine Name Source File Source Line Call Count
Board::BoardPaint Board.cs 329 8
Board::Go Board.cs 681 2
mainForm::btn_Play_Click MainForm.cs 588 1
mainForm::btnStart_Click MainForm.cs 614 1
Board::Initialize Board.cs 101 1
Board::Go Board.cs 681 1
Board::set_NumLines Board.cs 822 1
Board::.ctor Board.cs 71 1
mainForm::.ctor MainForm.cs 52 1
Board::DoOptions Board.cs 565 1
mainForm::.ctor MainForm.cs 52 1
Event View Disassembler Editor Details Call Graph Call Tree

In addition, a click on a routine or a class in the Report panel will refresh the contents of the **Details** panel. For routines, this panel represents the structure of function calls, but it does so as two lists. This lets you see all possible parents and children of the selected method. See *Static Analysis Profiler - Columns of the Details and Call Tree Panels*. Double-clicking a method line in either list (**Children** or **Parents**) refreshes Details and all other panels with the information on the chosen method.

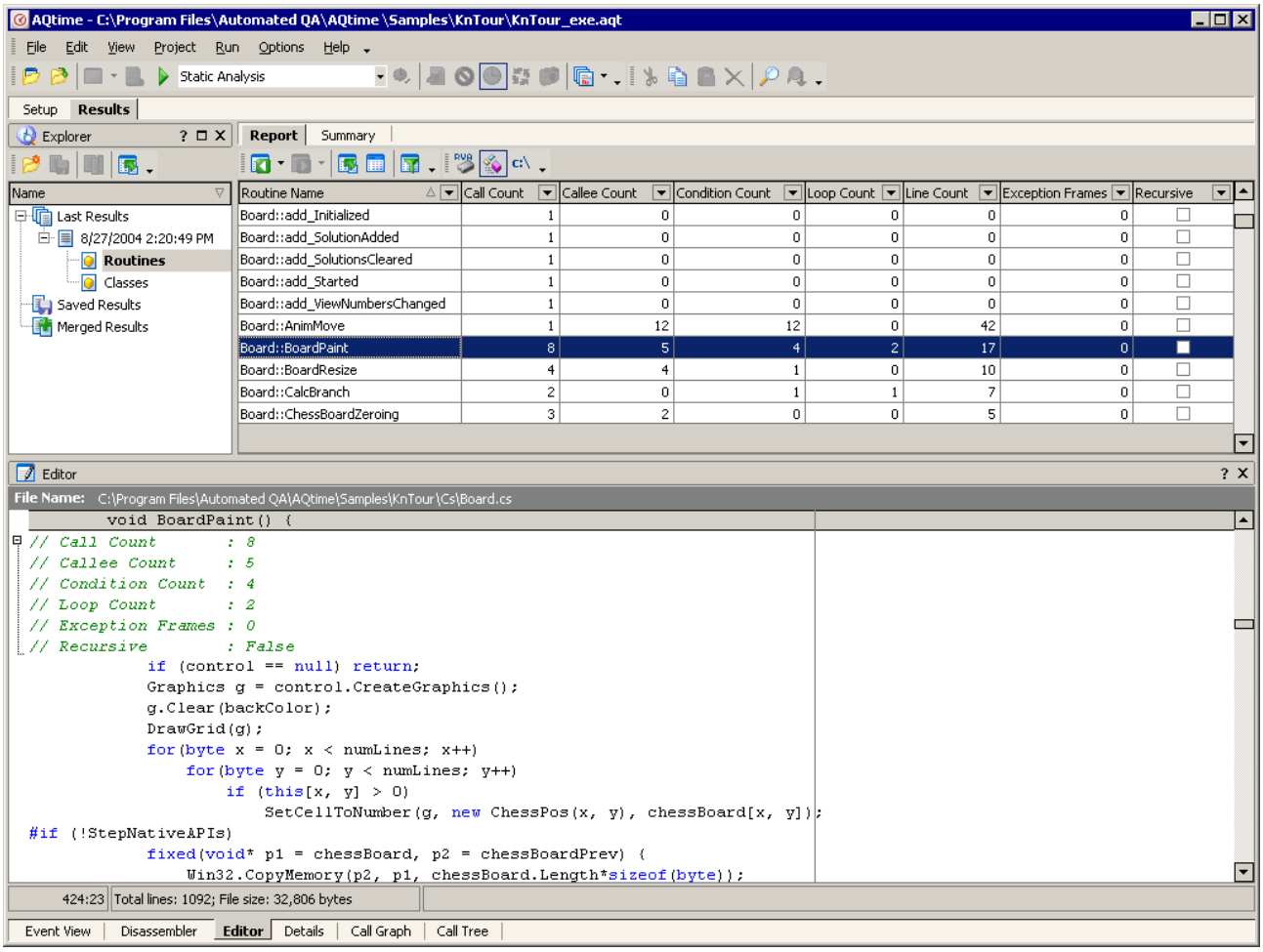
© 2010 AutomatedQA Corp.


AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 10:11:48 AM
Routines
Classes
Saved Results
Merged Results
Routine Name | Call Count | Callee Count | Condition Count | Loop Count | Line Count | Exception Frames | Recursive
Board::add_SolutionAdded | 1 | 0 | 0 | 0 | 0 | 0
Board::add_SolutionsCleared | 1 | 0 | 0 | 0 | 0 | 0
Board::add_Started | 1 | 0 | 0 | 0 | 0 | 0
Board::add_ViewNumbersChanged | 1 | 0 | 0 | 0 | 0 | 0
Board::AnimMove | 1 | 12 | 12 | 0 | 42 | 0
Board::BoardPaint | 8 | 5 | 4 | 2 | 17 | 0
Board::BoardResize | 4 | 4 | 1 | 0 | 10 | 0
Board::CalcBranch | 2 | 0 | 1 | 1 | 7 | 0
Board::ChessBoardZeroing | 3 | 2 | 0 | 0 | 5 | 0
Board::DoBoardResized | 1 | 1 | 0 | 0 | 2 | 0
Board::DoDrawBranchInfo | 2 | 1 | 0 | 0 | 2 | 0
Board::DoDrawBranchNum | 4 | 1 | 0 | 0 | 2 | 0
Board::DoDrawSolutions | 3 | 1 | 0 | 0 | 2 | 0
Board::DoOptions | 1 | 7 | 1 | 0 | 15 | 0
Board::DoSolutionAdded | 1 | 1 | 0 | 0 | 2 | 0
Board::DoSolutionsCleared | 2 | 1 | 0 | 0 | 2 | 0
Board::DrawAnimBitmap | 2 | 1 | 1 | 0 | 7 | 0
Board::DrawColorBitmap | 1 | 0 | 1 | 1 | 15 | 0
Board::DrawColorRect | 1 | 0 | 0 | 0 | 3 | 0
Board::DrawGrid | 1 | 1 | 4 | 3 | 21 | 0
Details
Parents Children
Routine Name | Source File | Source Line | Call Count
ChessPos::.ctor | Board.cs | 54 | 1
Board::DrawGrid | Board.cs | 155 | 1
Board::SetCellToNumber | Board.cs | 201 | 1
Board::get_Item | Board.cs | 746 | 1
Win32::CopyMemory | | 0 | 1
Children
Event View Disassembler Editor Details Call Graph Call Tree

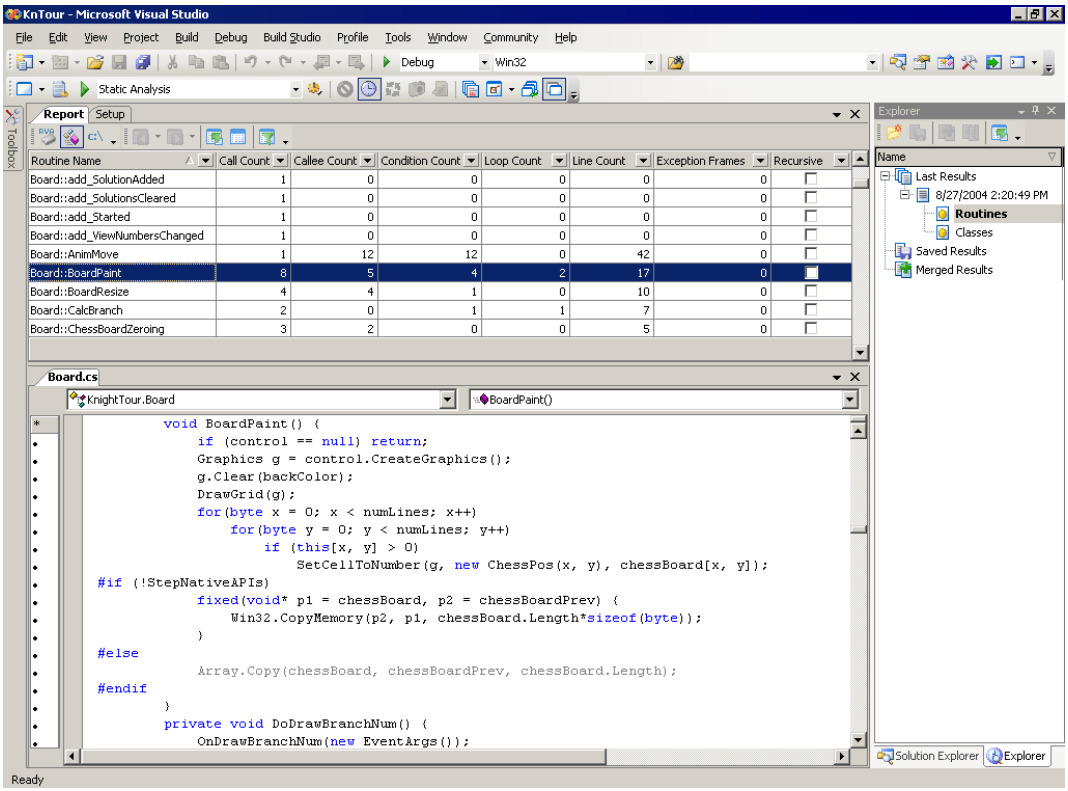
For classes, the Details panel displays information on class interlinks for the class chosen in the Report panel. This information is divided into several lists. The **Routines** list simply itemizes all methods of the given class. The **Class Callers** and **Class Callees** lists display the classes whose methods call methods of the given class or are called by methods of this class. If you click a class in either list, its methods will be displayed in the corresponding list **(Caller Routines** or **Callee Routines**). Double-clicking a class line in either list (Class Callers or Class Callees) refreshes Details and all other panels with the information on the chosen class. See *Static Analysis Profiler - Columns of the Details and Call Tree Panels*.



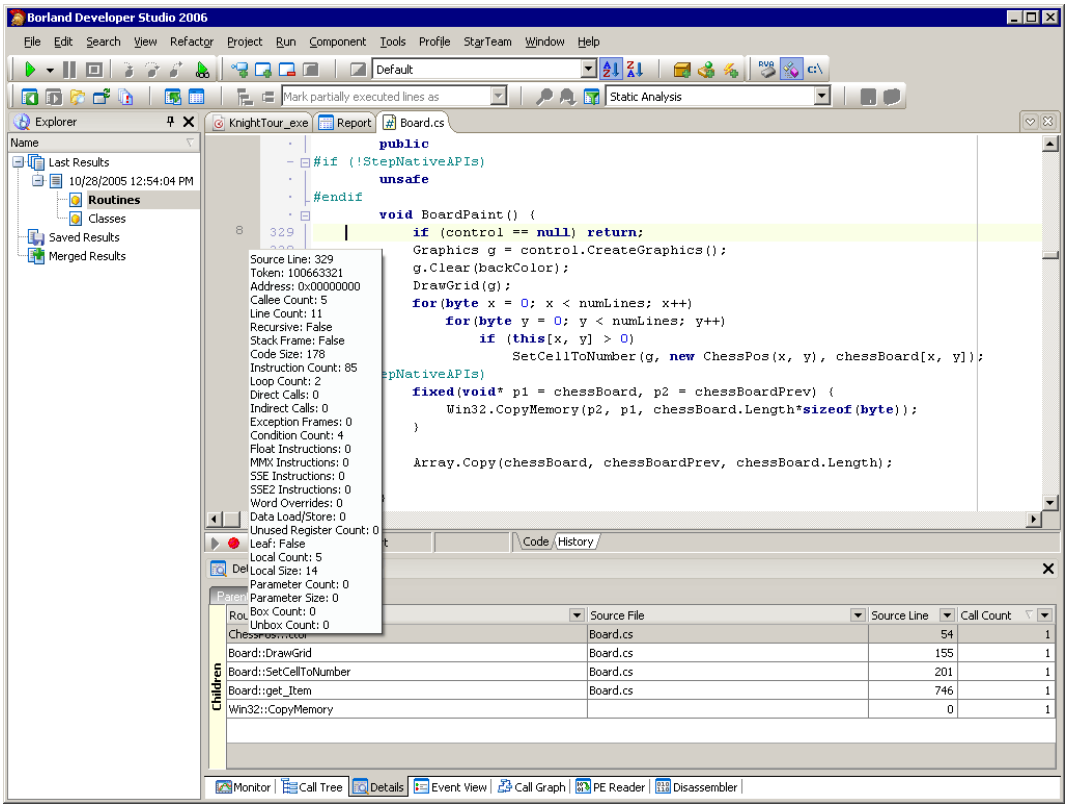
To view the source code of a routine, double-click it in the Report, Details, Call Graph or Call Tree panel and then switch to the **Editor** panel (which is available only if AQtime is running as a standalone application), to the **Code Editor** (which is the native editor of Microsoft Visual Studio) or to the **Editor** (which is the native editor of Borland Developer Studio). The path to source files must be specified in the Project Search Directories or Search Directories dialogs. In the Editor, the routine code will be accompanied with the routine's profiling results that are represented as comments.


AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 2:20:49 PM
Routines
Classes
Saved Results
Merged Results
Routine Name | Call Count | Callee Count | Condition Count | Loop Count | Line Count | Exception Frames | Recursive
Board::add_Initialized | 1 | 0 | 0 | 0 | 0 | 0
Board::add_SolutionAdded | 1 | 0 | 0 | 0 | 0 | 0
Board::add_SolutionsCleared | 1 | 0 | 0 | 0 | 0 | 0
Board::add_Started | 1 | 0 | 0 | 0 | 0 | 0
Board::add_ViewNumbersChanged | 1 | 0 | 0 | 0 | 0 | 0
Board::AnimMove | 1 | 12 | 12 | 0 | 42 | 0
Board::BoardPaint | 8 | 5 | 4 | 2 | 17 | 0
Board::BoardResize | 4 | 4 | 1 | 0 | 10 | 0
Board::CalcBranch | 2 | 0 | 1 | 1 | 7 | 0
Board::ChessBoardZeroing | 3 | 2 | 0 | 0 | 5 | 0
Editor
File Name: C:\Program Files\Automated QA\AQtime\Samples\KnTour\Cs\Board.cs
void BoardPaint() {
// Call Count : 8
// Callee Count : 5
// Condition Count : 4
// Loop Count : 2
// Exception Frames : 0
// Recursive : False
if (control == null) return;
Graphics g = control.CreateGraphics();
g.Clear(backColor);
DrawGrid(g);
for(byte x = 0; x < numLines; x++)
for(byte y = 0; y < numLines; y++)
if (this[x, y] > 0)
SetCellToNumber(g, new ChessPos(x, y), chessBoard[x, y]);
#if (!StepNativeAPIs)
fixed(void* p1 = chessBoard, p2 = chessBoardPrev) {
Win32.CopyMemory(p2, p1, chessBoard.Length*sizeof(byte));
424:23
Total lines: 1092; File size: 32,806 bytes
Event View Disassembler Editor Details Call Graph Call Tree

AQtime standalone



AQtime integrated into Visual Studio



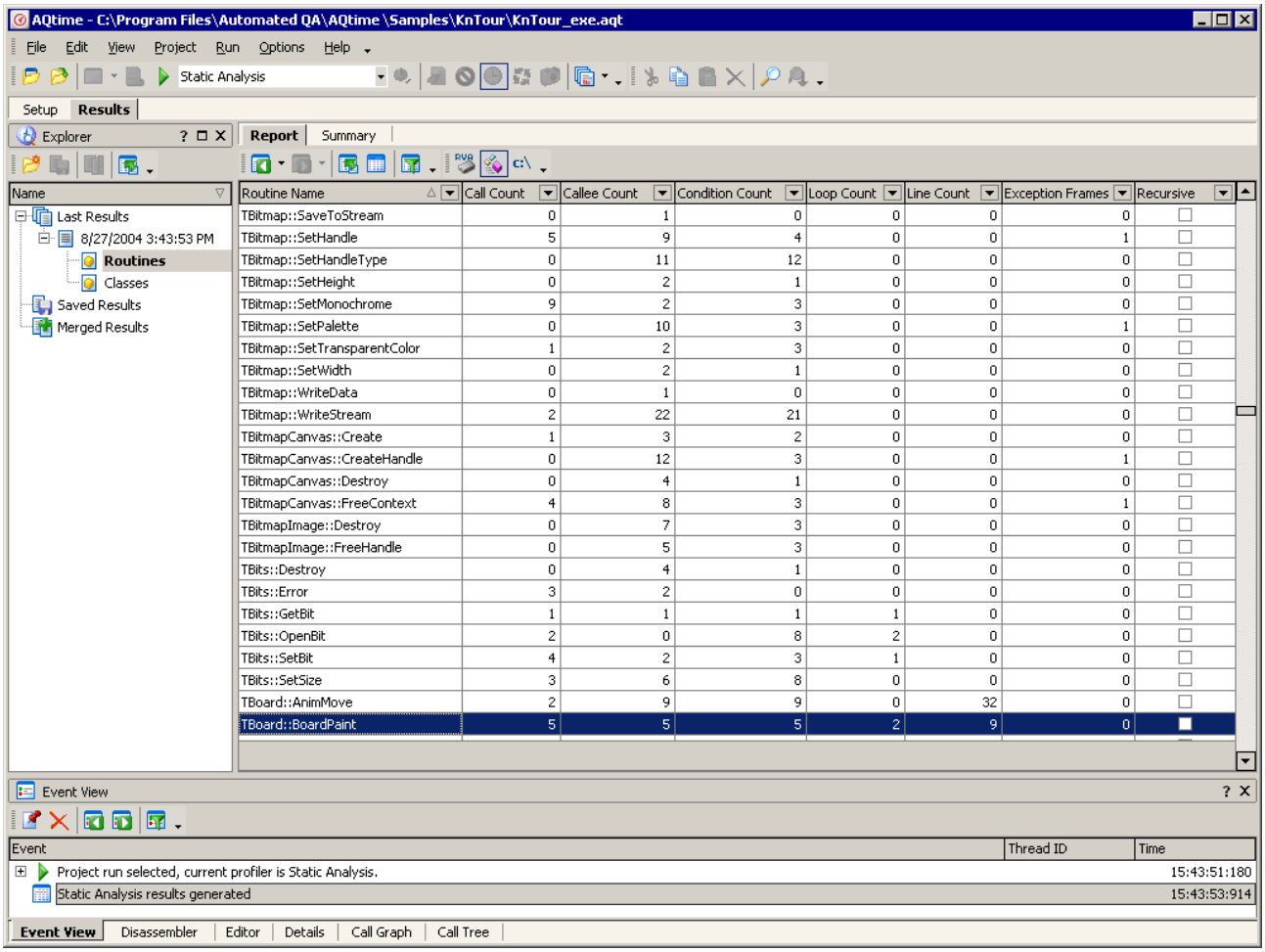
AQtime integrated into Borland Developer Studio

In Visual Studio, to view the routine's profiling results, right-click somewhere within the routine's code and select **Show Routine Summary** from the context menu, which will call the **Routine Summary** dialog.

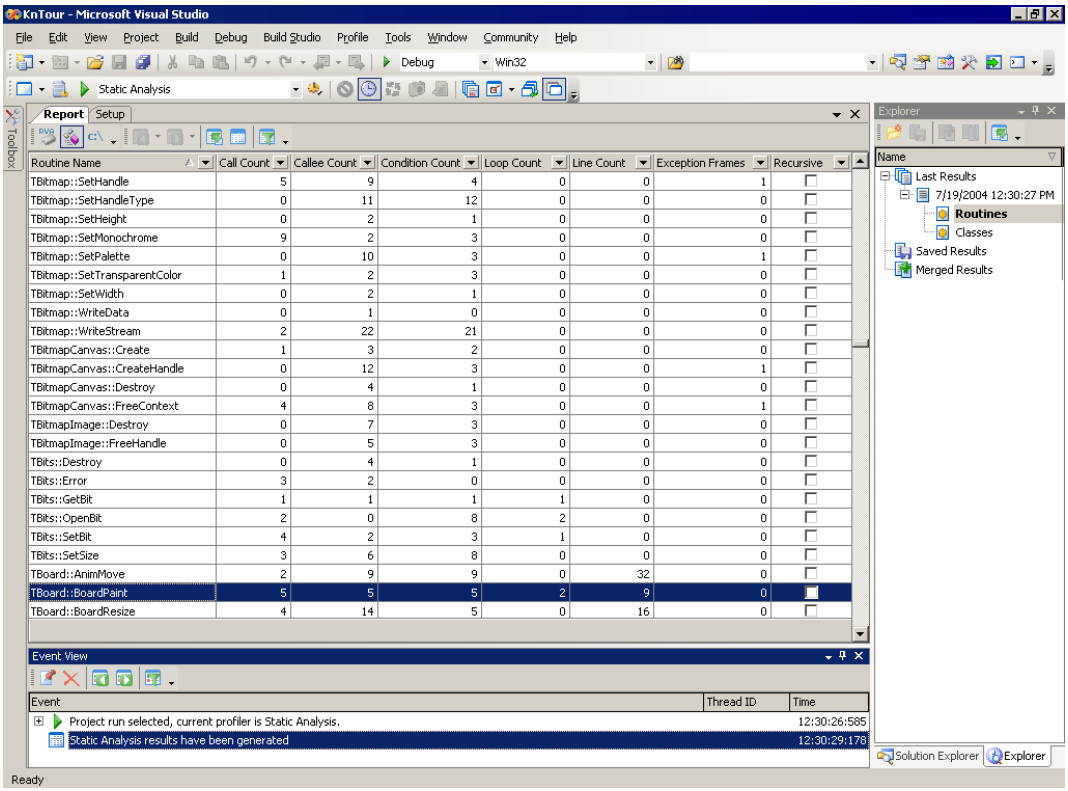
Static Analysis Profiler - Analyzing Results

The **Static Analysis** profiler lets you get the entire hierarchy of routine calls and class interlinks as they are coded in the application's source. The profiler does not even execute the application, it simply takes all these data from debugging information (for native applications) or metadata (for .NET-connected applications). The profiling results are organized into two categories: Routines and Classes. The **Routines** category contains results for each single routine that exists in the application being profiled. The **Classes** category allows you to view summary profiling results for each class in your application.

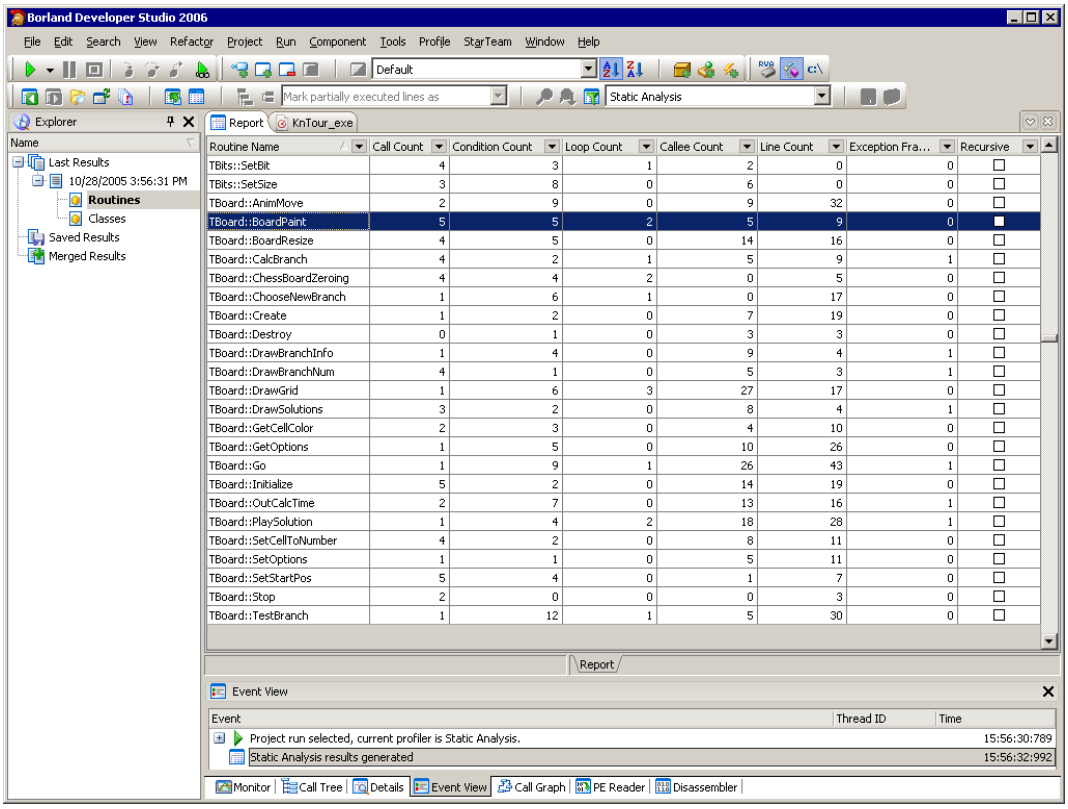
Here is a sample output of the Static Analysis profiler:


AQtime - C:\Program Files\Automated QA\AQtime\Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 3:43:53 PM
Routines
Classes
Saved Results
Merged Results
Routine Name | Call Count | Callee Count | Condition Count | Loop Count | Line Count | Exception Frames | Recursive
TBitmap::SaveToStream 0 1 0 0 0 0
TBitmap::SetHandle 5 9 4 0 0 1
TBitmap::SetHandleType 0 11 12 0 0 0
TBitmap::SetHeight 0 2 1 0 0 0
TBitmap::SetMonochrome 9 2 3 0 0 0
TBitmap::SetPalette 0 10 3 0 0 1
TBitmap::SetTransparentColor 1 2 3 0 0 0
TBitmap::SetWidth 0 2 1 0 0 0
TBitmap::WriteData 0 1 0 0 0 0
TBitmap::WriteStream 2 22 21 0 0 0
TBitmapCanvas::Create 1 3 2 0 0 0
TBitmapCanvas::CreateHandle 0 12 3 0 0 1
TBitmapCanvas::Destroy 0 4 1 0 0 0
TBitmapCanvas::FreeContext 4 8 3 0 0 1
TBitmapImage::Destroy 0 7 3 0 0 0
TBitmapImage::FreeHandle 0 5 3 0 0 0
TBits::Destroy 0 4 1 0 0 0
TBits::Error 3 2 0 0 0 0
TBits::GetBit 1 1 1 1 0 0
TBits::OpenBit 2 0 8 2 0 0
TBits::SetBit 4 2 3 1 0 0
TBits::SetSize 3 6 8 0 0 0
TBoard::AnimMove 2 9 9 0 32 0
TBoard::BoardPaint 5 5 5 2 9 0
Event View
Event | Thread ID | Time
Project run selected, current profiler is Static Analysis. 15:43:51:180
Static Analysis results generated 15:43:53:914
Event View Disassembler Editor Details Call Graph Call Tree

AQtime standalone


KnTour - Microsoft Visual Studio
File Edit View Project Build Debug Build Studio Profile Tools Window Community Help
Debug
Win32
Static Analysis
Report
Setup
Routine Name | Call Count | Callee Count | Condition Count | Loop Count | Line Count | Exception Frames | Recursive
TBitmap::SetHandle 5 9 4 0 0 1
TBitmap::SetHandleType 0 11 12 0 0 0
TBitmap::SetHeight 0 2 1 0 0 0
TBitmap::SetMonochrome 9 2 3 0 0 0
TBitmap::SetPalette 0 10 3 0 0 1
TBitmap::SetTransparentColor 1 2 3 0 0 0
TBitmap::SetWidth 0 2 1 0 0 0
TBitmap::WriteData 0 1 0 0 0 0
TBitmap::WriteStream 2 22 21 0 0 0
TBitmapCanvas::Create 1 3 2 0 0 0
TBitmapCanvas::CreateHandle 0 12 3 0 0 1
TBitmapCanvas::Destroy 0 4 1 0 0 0
TBitmapCanvas::FreeContext 4 8 3 0 0 1
TBitmapImage::Destroy 0 7 3 0 0 0
TBitmapImage::FreeHandle 0 5 3 0 0 0
TBits::Destroy 0 4 1 0 0 0
TBits::Error 3 2 0 0 0 0
TBits::GetBit 1 1 1 1 0 0
TBits::OpenBit 2 0 8 2 0 0
TBits::SetBit 4 2 3 1 0 0
TBits::SetSize 3 6 8 0 0 0
TBoard::AnimMove 2 9 9 0 32 0
TBoard::BoardPaint 5 5 5 2 9 0
TBoard::BoardResize 4 14 5 0 16 0
Event View
Event | Thread ID | Time
Project run selected, current profiler is Static Analysis. 12:30:26:585
Static Analysis results have been generated 12:30:29:178
Explorer
Name
Last Results
7/19/2004 12:30:27 PM
Routines
Classes
Saved Results
Merged Results
Solution Explorer
Explorer
Ready

AQtime integrated into Visual Studio


Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Default
Mark partially executed lines as
Static Analysis
Explorer
Name
Last Results
10/28/2005 3:56:31 PM
Routines
Classes
Saved Results
Merged Results
Report
KnTour_exe
Routine Name | Call Count | Condition Count | Loop Count | Callee Count | Line Count | Exception Fra... | Recursive
TBits::SetBit 4 3 1 2 0 0
TBits::SetSize 3 8 0 6 0 0
TBoard::AnimMove 2 9 0 9 32 0
TBoard::BoardPaint 5 5 2 5 9 0
TBoard::BoardResize 4 5 0 14 16 0
TBoard::CalcBranch 4 2 1 5 9 1
TBoard::ChessBoardZeroing 4 4 2 0 5 0
TBoard::ChooseNewBranch 1 6 1 0 17 0
TBoard::Create 1 2 0 7 19 0
TBoard::Destroy 0 1 0 3 3 0
TBoard::DrawBranchInfo 1 4 0 9 4 1
TBoard::DrawBranchNum 4 1 0 5 3 1
TBoard::DrawGrid 1 6 3 27 17 0
TBoard::DrawSolutions 3 2 0 8 4 1
TBoard::GetCellColor 2 3 0 4 10 0
TBoard::GetOptions 1 5 0 10 26 0
TBoard::Go 1 9 1 26 43 1
TBoard::Initialize 5 2 0 14 19 0
TBoard::OutCalcTime 2 7 0 13 16 1
TBoard::PlaySolution 1 4 2 18 28 1
TBoard::SetCellToNumber 4 2 0 8 11 0
TBoard::SetOptions 1 1 0 5 11 0
TBoard::SetStartPos 5 4 0 1 7 0
TBoard::Stop 2 0 0 0 3 0
TBoard::TestBranch 1 12 1 5 30 0
Report
Event View
Event | Thread ID | Time
Project run selected, current profiler is Static Analysis. 15:56:30:789
Static Analysis results generated 15:56:32:992
Monitor
Call Tree
Details
Event View
Call Graph
PE Reader
Disassembler

AQtime integrated into Borland Developer Studio

As you can see, the categories are shown in the **Explorer** panel. You can also select the desired category from the **Result Items** toolbar item (by default, this item is hidden):

The **Summary** panel displays the summary results for the whole profiler run regardless of the selected category. Use this panel to quickly find routines that can potentially perform poorly. The contents of other panels depend on the currently selected category:

- If you select the *Routines* category, AQtime will display profiling results one routine per line in the Report panel. The **Report** panel is the "main"results display. Other AQtime panels, such as **Details**, **Call Graph** or **Call Tree**, hold additional results for the routine selected in the Report panel.
- If you select the *Classes* category, AQtime will display profiling results one class per line in the Report panel. Again, the Report panel will serve as the main results display, while other panels will contain additional results for the class selected in the Report panel.

Profiling Results - Report Panel

The Report panel displays results for the category that is selected in the Explorer panel or in the *Result Items* list. For more information about the available columns, see *Static Analysis Profiler - Report Panel Columns*. Note that by default the Report panel shows only a shred of available columns. You can easily add more columns to the panel. For more information on this, see *Adding and Removing Columns*. You can arrange the columns in the panel as you desire: move columns, change column width, etc. For more information on this, see *Arranging Columns, Lines and Panels*.

The **Profiler** toolbar contains items that allow you to modify the results that are currently being displayed as your needs dictate. For example, the **Show addresses as RVA** toolbar item lets you choose the format the profiler should use to display addresses in the **Address** column of the Report panel. Another toolbar item, **Routine name with class name**, specifies whether the routine name should be preceded with the name of the class the routine belongs to. For more information on the toolbar items, see *Static Analysis Profiler - Options*.

The column footer shows summary results for the values displayed in that column. You can customize the summary type and summary format using the **Format Columns** dialog or the context menu that is brought up by right-clicking the column footer. For instance, you can select one of the five summary types (Sum, Count, Avg, Min, Max) or you can hide the summary for the column.

By default, the column summary is calculated for all rows displayed in the Report panel. However, if you select two or more rows, AQtime will recalculate the summary for the selected rows only.

If the Routines category is selected, you can find the routines that are called most often or have the greatest number of instructions. To do this, sort results by the **Call Count** or **Instruction Count** column correspondingly. If the Classes category is selected, you can find classes that have the greatest number of methods, classes whose methods are called the most often and classes that call methods of other classes the most often. To do this, sort results by the **Routine Count**, **Caller Class Count** or **Callee Class Count** column correspondingly.

Additionally, you can quickly filter out routines (classes) that call or do not call other methods (methods of other classes). To do this, use the predefined result views **Non-leaf routines (classes)** or **Leaf routines**

(classes) correspondingly. If you want to separately view native code routines and classes, or .NET code routines and classes, simply select the appropriate predefined view: **Native code routines and classes** or **.NET code routines and classes**.

You can select any of these views from the **Result Views** dropdown list on the Standard toolbar, or from the **View | Result Views** menu or from the **Result Views** dialog. To display it, choose **Profile | Result Views** from Visual Studio's menu or click **Result Views** on Borland Developer Studio's **View.AQtime** toolbar. See *Result Views*.

You can also group results by any column. Instead of grouping columns manually, you can use the **View by module** predefined view to display profiling results grouped by the **Module Name** column. When you group results by a column, besides "global"summaries shown at the footer of the panel, AQtime displays "local"summaries at the end of each group node. For instance, grouping results by the **Class Name** column helps you find the total number of instructions in all class methods (the summary for the Instruction Count column should be enabled). For more information on how to group, sort, filter and search for profiling results, see *Analyzing Profiler Results*.

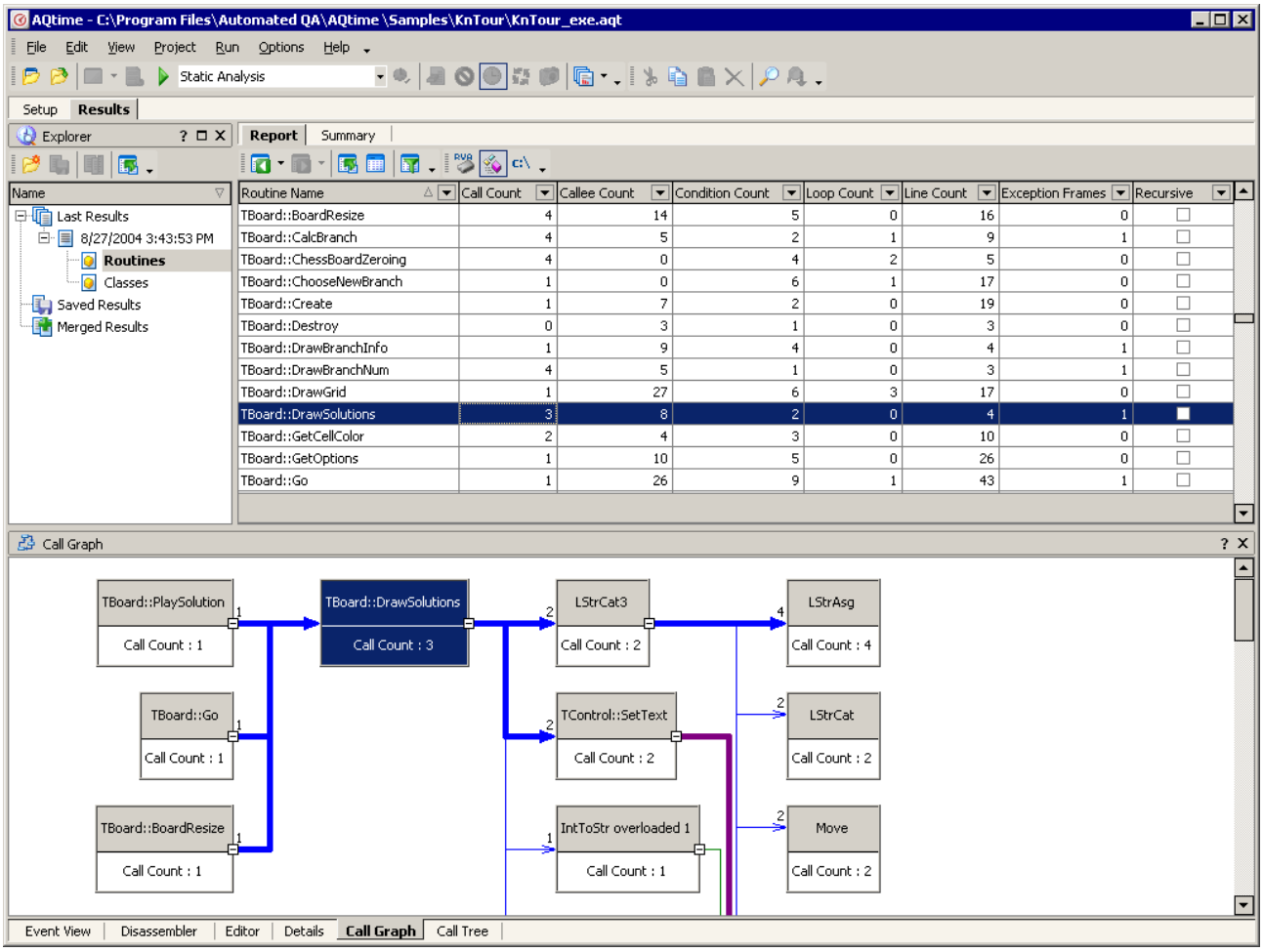
Suppose that you changed the source of your application, recompiled it and want to see how this modification affected the Static Analysis results. To do this, you can compare results of two profiler runs. See *Comparing and Merging Results*.

Profiling Results - Call Graph, Call Tree, Editor and Details Panels

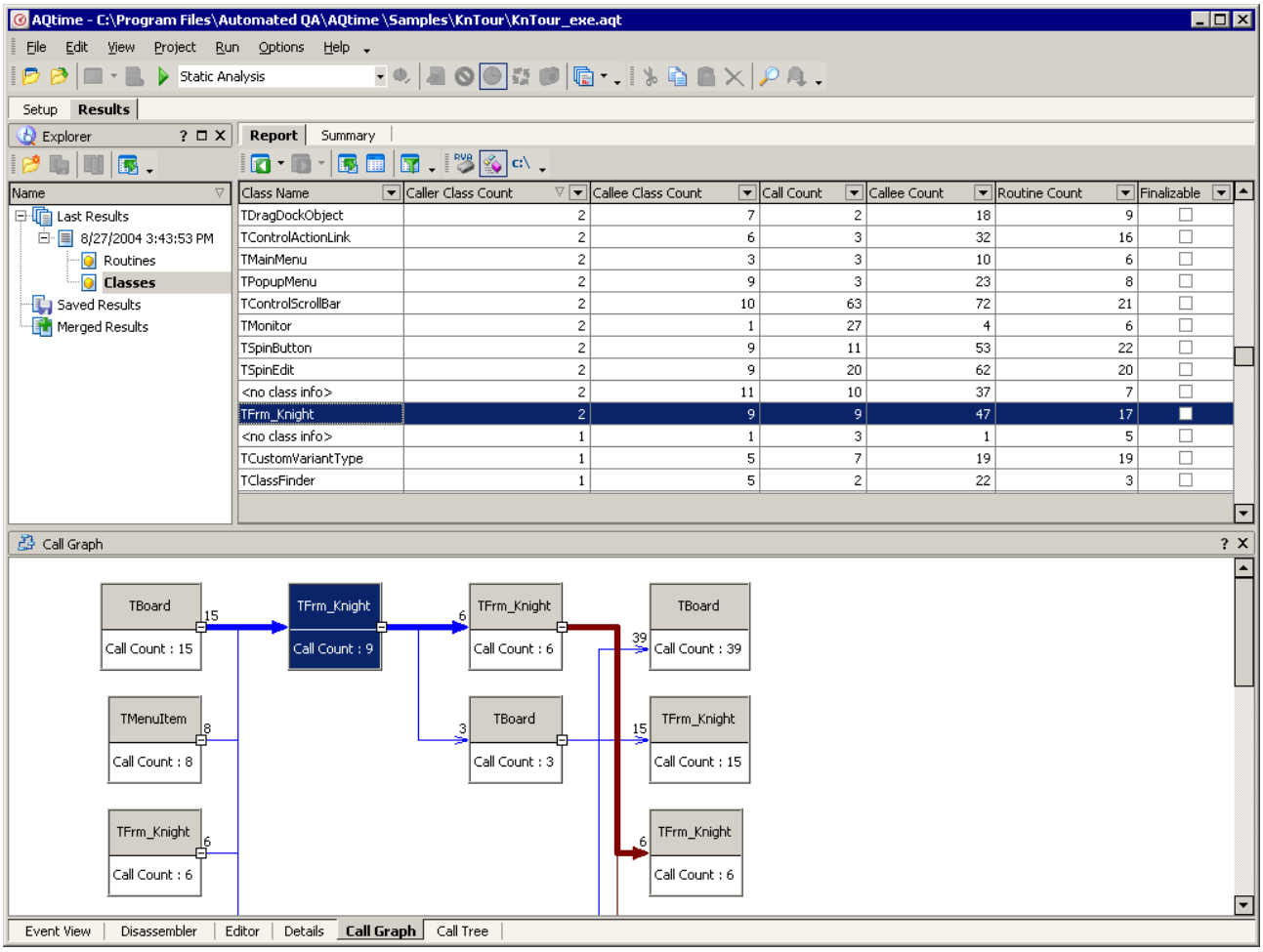
The **Static Analysis** profiler displays additional profiling results in the **Call Graph**, **Call Tree**, **Details** and **Editor** panels. When you double-click on a routine (a class) in Report, AQtime refreshes these panels so that they will display information about this routine (class). See *AQtime Panels*.

For routines, the Call Graph displays the hierarchy of potential function calls (parent - child), centering on the clicked method. The critical path for the routine is displayed in bold (critical path is the "longest"path for the routine in the hierarchy of potential function calls, for instance, the one that has the greatest number of instructions). To configure, value of which column will be used to calculate the critical path, use the **Customize Call Graph** dialog.

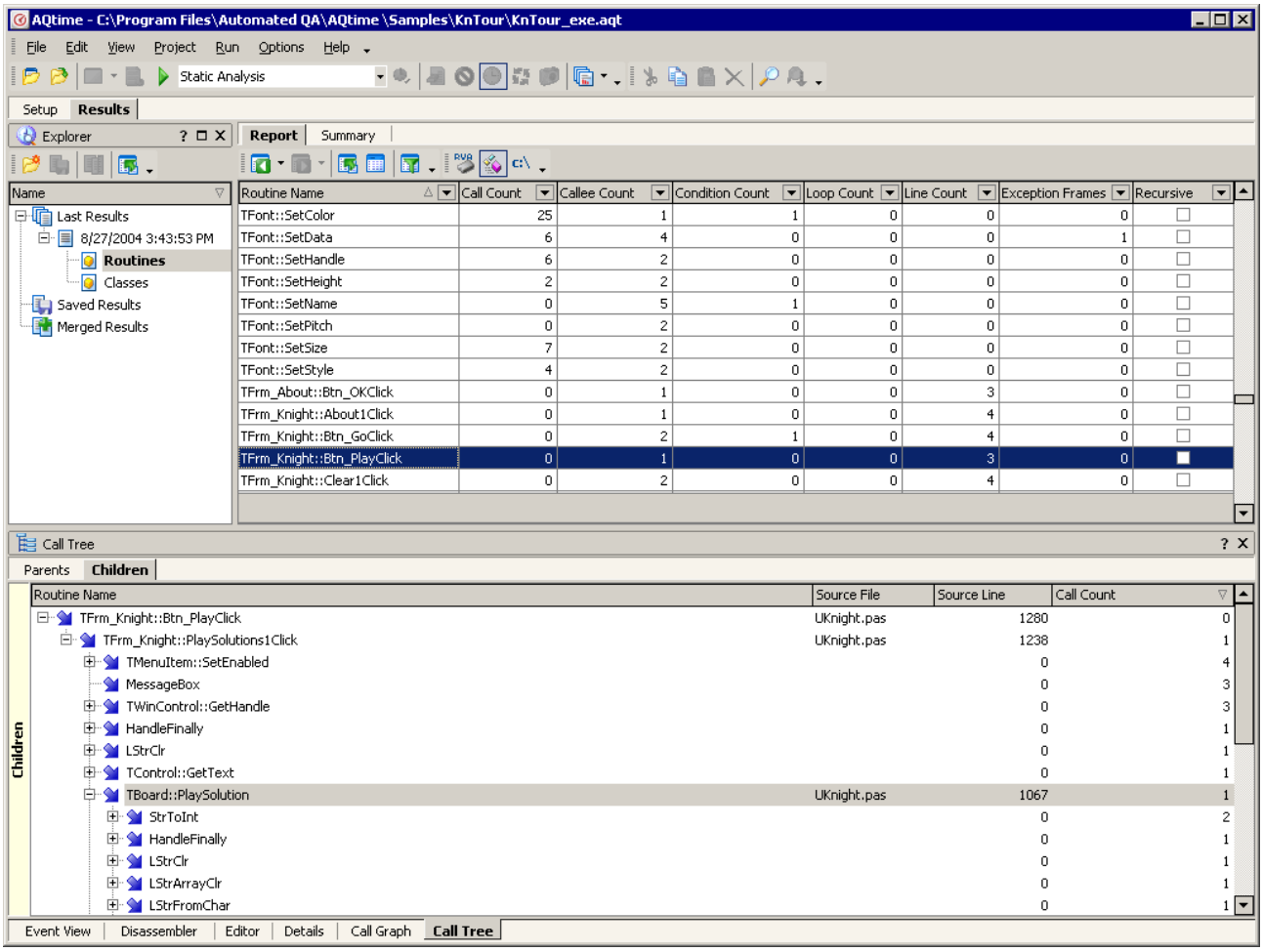
© 2010 AutomatedQA Corp.


AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 3:43:53 PM
Routines
Classes
Saved Results
Merged Results
Routine Name | Call Count | Callee Count | Condition Count | Loop Count | Line Count | Exception Frames | Recursive
TBoard::BoardResize 4 14 5 0 16 0
TBoard::CalcBranch 4 5 2 1 9 1
TBoard::ChessBoardZeroing 4 0 4 2 5 0
TBoard::ChooseNewBranch 1 0 6 1 17 0
TBoard::Create 1 7 2 0 19 0
TBoard::Destroy 0 3 1 0 3 0
TBoard::DrawBranchInfo 1 9 4 0 4 1
TBoard::DrawBranchNum 4 5 1 0 3 1
TBoard::DrawGrid 1 27 6 3 17 0
TBoard::DrawSolutions 3 8 2 0 4 1
TBoard::GetCellColor 2 4 3 0 10 0
TBoard::GetOptions 1 10 5 0 26 0
TBoard::Go 1 26 9 1 43 1
Call Graph
TBoard::PlaySolution
Call Count : 1
TBoard::DrawSolutions
Call Count : 3
LStrCat3
Call Count : 2
LStrAsg
Call Count : 4
TBoard::Go
Call Count : 1
TControl::SetText
Call Count : 2
LStrCat
Call Count : 2
TBoard::BoardResize
Call Count : 1
IntToStr overloaded 1
Call Count : 1
Move
Call Count : 2
Event View Disassembler Editor Details Call Graph Call Tree

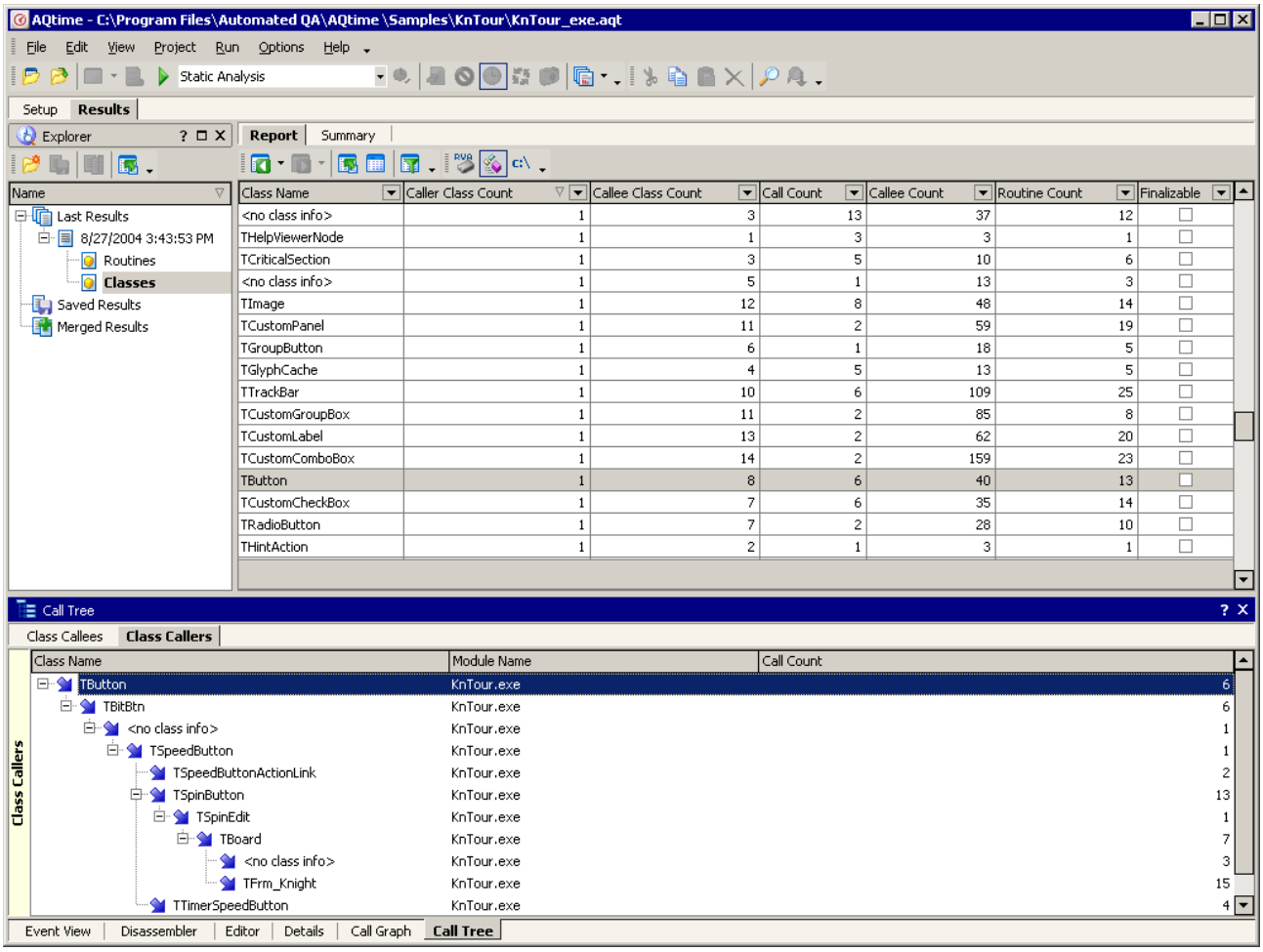
For classes, the **Call Graph** displays the hierarchy of potential links between classes through their methods. You can travel up and down the hierarchy in the panel by clicking.


AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 3:43:53 PM
Routines
Classes
Saved Results
Merged Results
Class Name | Caller Class Count | Callee Class Count | Call Count | Callee Count | Routine Count | Finalizable
TDragDockObject 2 7 2 18 9
TControlActionLink 2 6 3 32 16
TMainMenu 2 3 3 10 6
TPopupMenu 2 9 3 23 8
TControlScrollBar 2 10 63 72 21
TMonitor 2 1 27 4 6
TSpinButton 2 9 11 53 22
TSpinEdit 2 9 20 62 20
<no class info> 2 11 10 37 7
TFrm_Knight 2 9 9 47 17
<no class info> 1 1 3 1 5
TCustomVariantType 1 5 7 19 19
TClassFinder 1 5 2 22 3
Call Graph
TBoard Call Count : 15
TFrm_Knight Call Count : 9
TFrm_Knight Call Count : 6
TBoard Call Count : 39
TMenuItem Call Count : 8
TBoard Call Count : 3
TFrm_Knight Call Count : 15
TFrm_Knight Call Count : 6
TFrm_Knight Call Count : 6
Event View Disassembler Editor Details Call Graph Call Tree

For the Routines category, the **Call Tree** panel also displays the hierarchy of potential function calls. It contains two panes: Parents and Children. The **Parents** pane holds all function calls that lead to the call to the currently selected routine. The **Children** pane displays the hierarchy of child calls that are initiated from the selected routine.

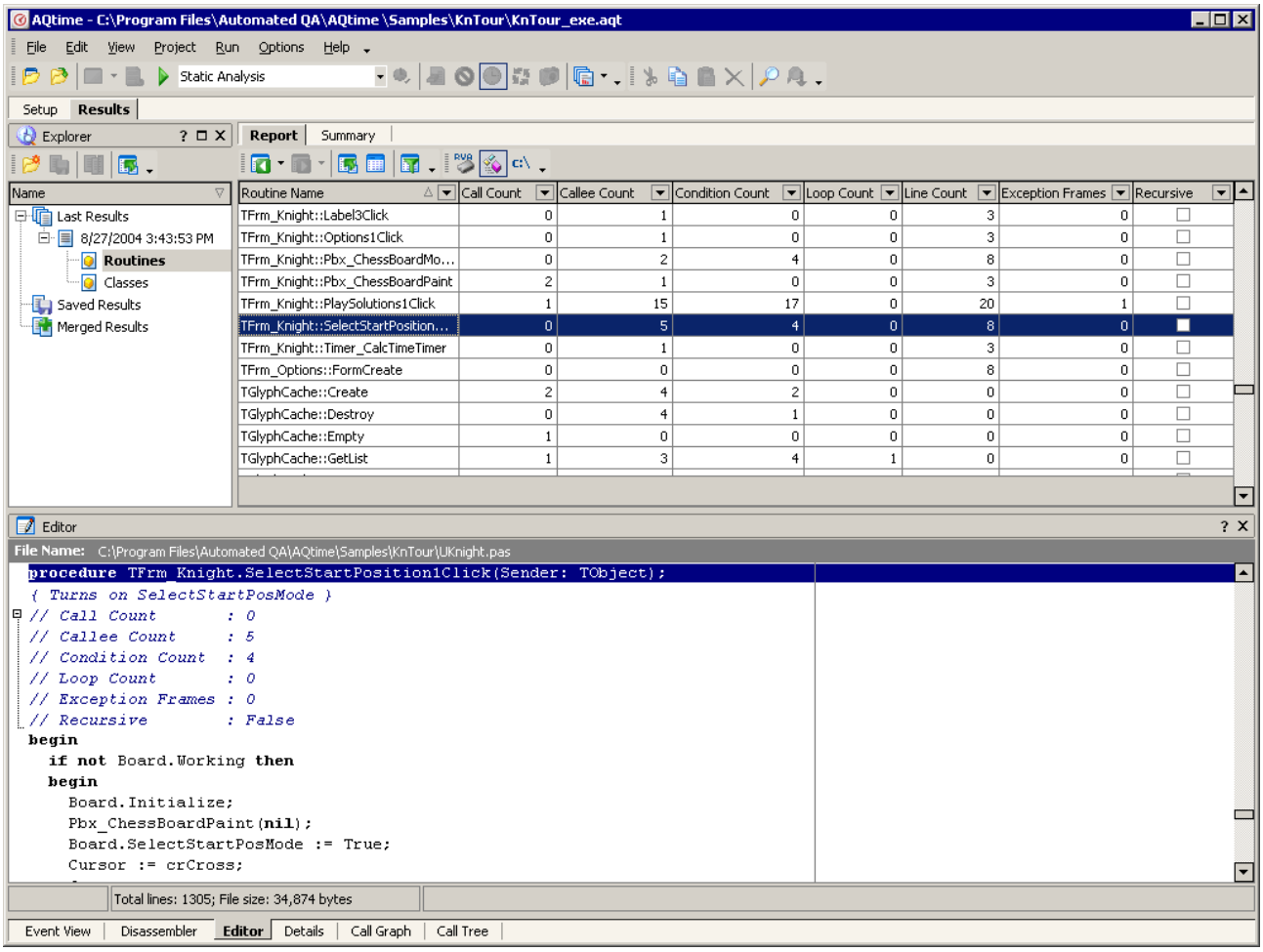

AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 3:43:53 PM
Routines
Classes
Saved Results
Merged Results
Routine Name | Call Count | Callee Count | Condition Count | Loop Count | Line Count | Exception Frames | Recursive
TFont::SetColor 25 1 1 0 0 0
TFont::SetData 6 4 0 0 0 1
TFont::SetHandle 6 2 0 0 0 0
TFont::SetHeight 2 2 0 0 0 0
TFont::SetName 0 5 1 0 0 0
TFont::SetPitch 0 2 0 0 0 0
TFont::SetSize 7 2 0 0 0 0
TFont::SetStyle 4 2 0 0 0 0
TFrm_About::Btn_OKClick 0 1 0 0 3 0
TFrm_Knight::About1Click 0 1 0 0 4 0
TFrm_Knight::Btn_GoClick 0 2 1 0 4 0
TFrm_Knight::Btn_PlayClick 0 1 0 0 3 0
TFrm_Knight::Clear1Click 0 2 0 0 4 0
Call Tree
Parents Children
Routine Name | Source File | Source Line | Call Count
TFrm_Knight::Btn_PlayClick UKnight.pas 1280 0
TFrm_Knight::PlaySolutions1Click UKnight.pas 1238 1
TMenuItem::SetEnabled 0 4
MessageBox 0 3
TWinControl::GetHandle 0 3
HandleFinally 0 1
LStrClr 0 1
TControl::GetText 0 1
TBoard::PlaySolution UKnight.pas 1067 1
StrToInt 0 2
HandleFinally 0 1
LStrClr 0 1
LStrArrayClr 0 1
LStrFromChar 0 1
Children
Event View Disassembler Editor Details Call Graph Call Tree

For the Classes category, the Call Tree panel shows the hierarchy of class interlinks. It contains two panes: Class Callers and Class Callees. The **Class Callers** pane holds the hierarchy of classes whose methods eventually call methods of the currently selected class. The **Class Callees** pane displays the hierarchy of classes calls to whose methods are initiated from the selected class.

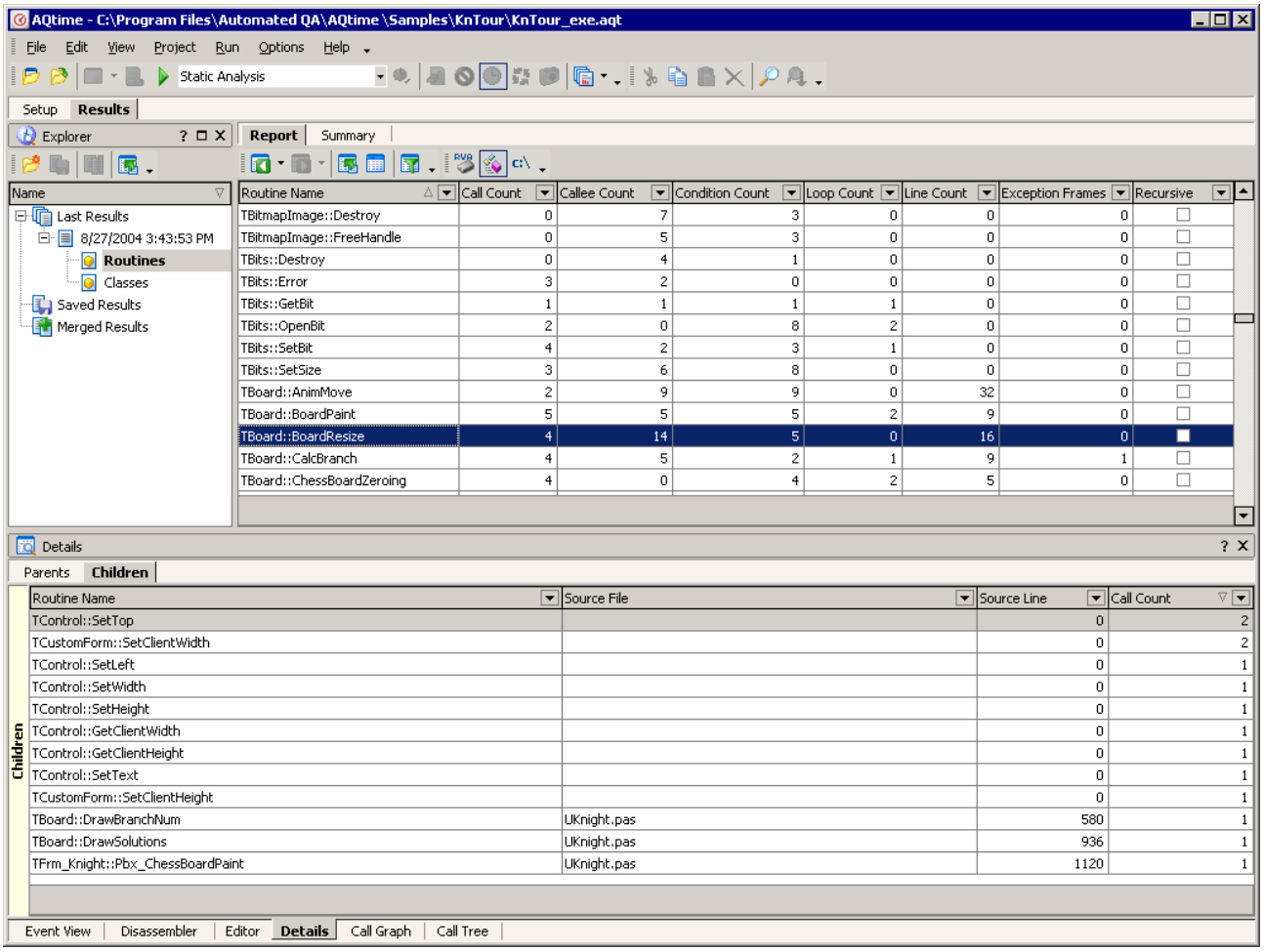

AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 3:43:53 PM
Routines
Classes
Saved Results
Merged Results
Class Name | Caller Class Count | Callee Class Count | Call Count | Callee Count | Routine Count | Finalizable
<no class info> 1 3 13 37 12
THelpViewerNode 1 1 3 3 1
TCriticalSection 1 3 5 10 6
<no class info> 1 5 1 13 3
TImage 1 12 8 48 14
TCustomPanel 1 11 2 59 19
TGroupButton 1 6 1 18 5
TGlyphCache 1 4 5 13 5
TTrackBar 1 10 6 109 25
TCustomGroupBox 1 11 2 85 8
TCustomLabel 1 13 2 62 20
TCustomComboBox 1 14 2 159 23
TButton 1 8 6 40 13
TCustomCheckBox 1 7 6 35 14
TRadioButton 1 7 2 28 10
THintAction 1 2 1 3 1
Call Tree
Class Callees Class Callers
Class Callers
Class Name | Module Name | Call Count
TButton KnTour.exe 6
TBitBtn KnTour.exe 6
<no class info> KnTour.exe 1
TSpeedButton KnTour.exe 1
TSpeedButtonActionLink KnTour.exe 2
TSpinButton KnTour.exe 13
TSpinEdit KnTour.exe 1
TBoard KnTour.exe 7
<no class info> KnTour.exe 3
TFrm_Knight KnTour.exe 15
TTimerSpeedButton KnTour.exe 4
Event View Disassembler Editor Details Call Graph Call Tree

For more information on results displayed in the Call Tree panel's panes as well as for the column description, see *Static Analysis Profiler - Call Tree Panel Columns*.

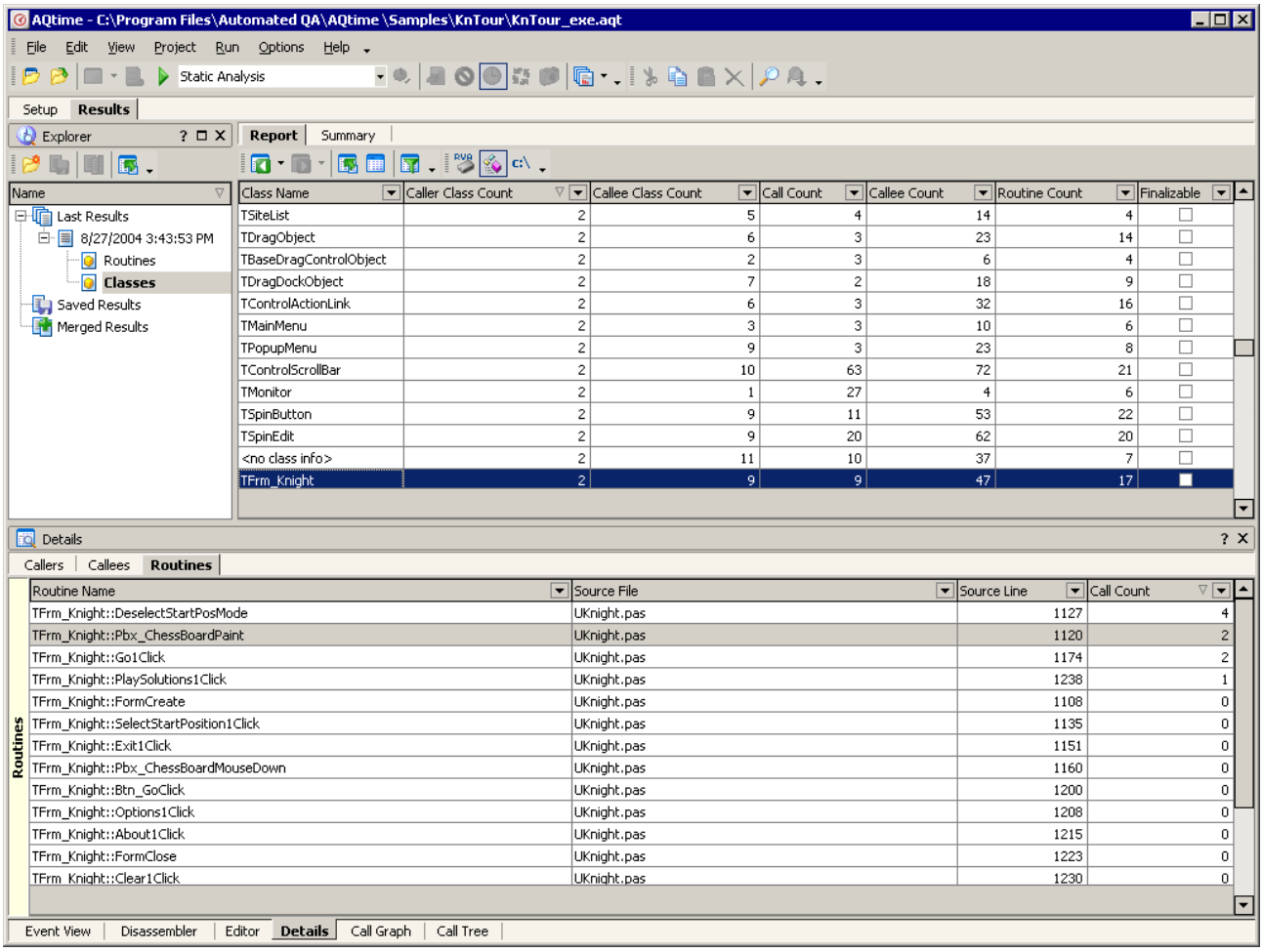
If your application was compiled with debug info, the **Editor** panel will also show the source code for the routine you clicked (The source file of the routine must be specified in the Search Directories or Project Search Directories). The Editor displays various profiling results as comments before the routine's source code. To configure which columns you want to see in these comments, use the **Customize Comments** dialog.


AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 3:43:53 PM
Routines
Classes
Saved Results
Merged Results
Routine Name | Call Count | Callee Count | Condition Count | Loop Count | Line Count | Exception Frames | Recursive
TFrm_Knight::Label3Click 0 1 0 0 3 0
TFrm_Knight::Options1Click 0 1 0 0 3 0
TFrm_Knight::Pbx_ChessBoardMo... 0 2 4 0 8 0
TFrm_Knight::Pbx_ChessBoardPaint 2 1 0 0 3 0
TFrm_Knight::PlaySolutions1Click 1 15 17 0 20 1
TFrm_Knight::SelectStartPosition... 0 5 4 0 8 0
TFrm_Knight::Timer_CalcTimeTimer 0 1 0 0 3 0
TFrm_Options::FormCreate 0 0 0 0 8 0
TGlyphCache::Create 2 4 2 0 0 0
TGlyphCache::Destroy 0 4 1 0 0 0
TGlyphCache::Empty 1 0 0 0 0 0
TGlyphCache::GetList 1 3 4 1 0 0
Editor
File Name: C:\Program Files\Automated QA\AQtime\Samples\KnTour\UKnight.pas
procedure TFrm_Knight.SelectStartPosition1Click(Sender: TObject);
{ Turns on SelectStartPosMode }
// Call Count : 0
// Callee Count : 5
// Condition Count : 4
// Loop Count : 0
// Exception Frames : 0
// Recursive : False
begin
if not Board.Working then
begin
Board.Initialize;
Pbx_ChessBoardPaint(nil);
Board.SelectStartPosMode := True;
Cursor := crCross;
Total lines: 1305; File size: 34,874 bytes
Event View Disassembler Editor Details Call Graph Call Tree

For the Routines category, the **Details** panel acts as a “magnifier”for parent-child call relationships related to one row in the Report panel. Routines that call the currently selected routine are listed in the **Parents** pane, while routines that are called by the selected routine are listed in the **Children** pane.


AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 3:43:53 PM
Routines
Classes
Saved Results
Merged Results
Routine Name | Call Count | Callee Count | Condition Count | Loop Count | Line Count | Exception Frames | Recursive
TBitmapImage::Destroy 0 7 3 0 0 0
TBitmapImage::FreeHandle 0 5 3 0 0 0
TBits::Destroy 0 4 1 0 0 0
TBits::Error 3 2 0 0 0 0
TBits::GetBit 1 1 1 1 0 0
TBits::OpenBit 2 0 8 2 0 0
TBits::SetBit 4 2 3 1 0 0
TBits::SetSize 3 6 8 0 0 0
TBoard::AnimMove 2 9 9 0 32 0
TBoard::BoardPaint 5 5 5 2 9 0
TBoard::BoardResize 4 14 5 0 16 0
TBoard::CalcBranch 4 5 2 1 9 1
TBoard::ChessBoardZeroing 4 0 4 2 5 0
Details
Parents Children
Routine Name | Source File | Source Line | Call Count
TControl::SetTop 0 2
TCustomForm::SetClientWidth 0 2
TControl::SetLeft 0 1
TControl::SetWidth 0 1
TControl::SetHeight 0 1
TControl::GetClientWidth 0 1
TControl::GetClientHeight 0 1
TControl::SetText 0 1
TCustomForm::SetClientHeight 0 1
TBoard::DrawBranchNum UKnight.pas 580 1
TBoard::DrawSolutions UKnight.pas 936 1
TFrm_Knight::Pbx_ChessBoardPaint UKnight.pas 1120 1
Children
Event View Disassembler Editor Details Call Graph Call Tree

For the Classes category, the Details panel displays additional information on the class selected in the Report panel. The **Routines** pane displays all methods of the class that is currently selected in the Report panel. The **Class Callers** pane lists classes whose methods call methods of the current class. At that, the **Caller Routines** displays all methods of the class chosen in Class Callers. Similarly, the **Class Callees** pane lists classes whose methods are called by methods of the current class, while the **Callee Routines** pane shows all methods of the class chosen in Class Callees.


AQtime - C:\Program Files\Automated QA\AQtime \Samples\KnTour\KnTour_exe.aqt
File Edit View Project Run Options Help
Static Analysis
Setup Results
Explorer
Report Summary
Name
Last Results
8/27/2004 3:43:53 PM
Routines
Classes
Saved Results
Merged Results
Class Name | Caller Class Count | Callee Class Count | Call Count | Callee Count | Routine Count | Finalizable
TSiteList 2 5 4 14 4
TDragObject 2 6 3 23 14
TBaseDragControlObject 2 2 3 6 4
TDragDockObject 2 7 2 18 9
TControlActionLink 2 6 3 32 16
TMainMenu 2 3 3 10 6
TPopupMenu 2 9 3 23 8
TControlScrollBar 2 10 63 72 21
TMonitor 2 1 27 4 6
TSpinButton 2 9 11 53 22
TSpinEdit 2 9 20 62 20
<no class info> 2 11 10 37 7
TFrm_Knight 2 9 9 47 17
Details
Callers Callees Routines
Routines
Routine Name | Source File | Source Line | Call Count
TFrm_Knight::DeselectStartPosMode UKnight.pas 1127 4
TFrm_Knight::Pbx_ChessBoardPaint UKnight.pas 1120 2
TFrm_Knight::Go1Click UKnight.pas 1174 2
TFrm_Knight::PlaySolutions1Click UKnight.pas 1238 1
TFrm_Knight::FormCreate UKnight.pas 1108 0
TFrm_Knight::SelectStartPosition1Click UKnight.pas 1135 0
TFrm_Knight::Exit1Click UKnight.pas 1151 0
TFrm_Knight::Pbx_ChessBoardMouseDown UKnight.pas 1160 0
TFrm_Knight::Btn_GoClick UKnight.pas 1200 0
TFrm_Knight::Options1Click UKnight.pas 1208 0
TFrm_Knight::About1Click UKnight.pas 1215 0
TFrm_Knight::FormClose UKnight.pas 1223 0
TFrm_Knight::Clear1Click UKnight.pas 1230 0
Event View Disassembler Editor Details Call Graph Call Tree

For more information on results displayed in the Details panel's panes as well as for the column description, see *Static Analysis Profiler - Details Panel Columns*.

Double-clicking on a row in the Details panel will move the cursor to the Editor panel to the routine (or class) displayed on that row. Also, the double-click will update the other panels to the clicked routine or class. Switching from panel to panel in this way, when trying to get the desired information out of the Static Analysis profiler results, is made much easier by the “browser”buttons, **Display Previous** and **Display Next**, on the Report toolbar.

You can arrange the panes of the Details panel as you desire. For more information on this, see description of the **Details** panel.

Static Analysis Profiler - Report Panel Columns

When you view results of the **Static Analysis** profiler, the **Report** panel contents depend on the currently selected category in the **Explorer** panel.

Classes Category

When this category is active, the Report panel shows information about classes whose instances can be potentially created in the profiled application. The panel holds the following columns:

Columns (in alphabetical order)	**Description**
Call Count	The number of calls to the class methods in the source.

Callee Class Count	The number of classes whose methods are called by methods of the given class.
Callee Count	The number of calls to other routines coded in the class source.
Caller Class Count	The number of classes whose methods call methods of the given class.
Class Name	Name of the class.
Finalizable	Specifies whether the class overrides the `Finalize` method (C# and VC++.NET use the destructor syntax for `Finalize`).
Module Name	Name of the executable module where the class is defined.
Namespace	Name of the namespace to which the class belongs. This column is only used for managed routines.
Routine Count	The number of methods in the class.
Token	CLR token of the class.

Routines Category

When this category is active, the Report panel shows information about routines that can be potentially called in the profiled application. The panel holds the following columns:

Columns (in alphabetical order)	**Description**
Address	The routine's address in memory. This column is only used for unmanaged (native-code) routines. The format of the address depends upon the *Show Addresses as RVA* option.
Box Count	The number of times the routine is boxed in the source.
Call Count	The number of calls to the routine coded in the source.
Callee Count	The number of calls to other routines coded in the source of the given routine.
Class Name	Name of the class to which the given routine belongs.
Code Size	Size of the routine's binary code (in bytes).
Code Type	Type of the routine's code. Possible values are *x86*, *MSIL* or *PInvoke*.
Condition Count	The number of conditional instructions in the routine's source.
Data Load/Store	The number of memory handling instructions in the routine's source.
Direct Calls	The number of direct calls to other routines performed in the routine's source.
Exception Frames	The number of exception frames in the routine's source.
Float Instructions	The number of floating point instructions in the routine's source.
Indirect Calls	The number of indirect calls to other routines performed in the routine's source. These are calls to callback functions (when a pointer to a routine is passed somewhere where the routine is actually called) and calls to interface functions.
Instruction Count	The total number of instructions in the routine's code.
Leaf	Specifies whether the routine is leaf or not. Leaf routines are those that do not call other routines.
Line Count	The number of lines in the routine's source.

Local Count	The number of local variables in the routine's source
Local Size	The size of memory (in bytes) occupied by the routine's local variables.
Loop Count	The number of loop instructions in the routine's source.
MMX Instructions	The number of MMX instructions in the routine's source.
Module Name	Name of the executable module where the routine is defined.
Namespace	Name of the namespace to which the routine's class belongs. This column is only used for managed routines.
Parameter Count	The number of parameters that are passed to the routine.
Parameter Size	The size of memory (in bytes) occupied by parameters that are passed to the routine.
Platform	The minimal processor configuration at which the routine's code can be run. This is determined by instructions used in the code. Possible values are *Blended* (means that the process type is not strictly defined; actually it means any Intel processor starting from Pentium), *Pentium II*, *Pentium III* or *Pentium IV*.
Routine Name	Name of the given routine.
Recursive	Specifies whether the routine has recursive calls (i.e. calls to itself).
Source File	Name of the source file for the routine.
Source Line	The source files line number where the routine's implementation begins.
SSE Instructions	The number of SSE instructions in the routine's source.
SSE2 Instructions	The number of SSE2 instructions in the routine's source.
Stack Frame	Specifies whether a stack frame is created for the given routine. Stack frames make debugging easier but hamper performance, because they involve execution of additional code.
Token	The routine's CLR token.
Unbox Count	The number of times the routine is unboxed in the source.
Unit Name	Name of the compiled linkage unit. This column is used for unmanaged (native-code) routines only.
Unused Register Count	The number of registers that are not used in the routine's source.
Word Overrides	The number of times when registers are used partly, not entirely. Large values of this counter lead to poor performance of the routine.

Static Analysis Profiler - Columns of the Details and Call Tree Panels

When you review the **Static Analysis** profiler results, the **Report** panel displays information on potential routine calls or potential calls between methods of the application's classes. The results that are shown in the Report, Details, Call Graph and Call Tree panels depend on the category selected in the **Explorer** panel.

Routines Category

If the *Routines* category is selected, the Report panel displays results for routines and the **Details** and **Call Tree** panels hold additional information about routine calls for the routine that is currently selected in the Report panel. Both the Details and Call Tree panels contain two panes: Parents and Children. The **Parents** pane lists all routines that call the currently selected routine in the application source. The **Children** pane

shows routines that are called by the selected routine. This information is shown in grids that hold the following columns (both Call Tree and Details have the same set of columns):

Columns (in alphabetical order)	**Description**
Call Count	The number of times the routine that is selected in Report called the given child routine or is called by the given parent routine.
Class Name	Name of the class to which the given routine belongs.
Code Type	Type of the routine's code. Possible values are *x86*, *MSIL* or *PInvoke*.
Module Name	Name of the executable module where the routine is defined.
Namespace	Name of the namespace to which the routine's class belongs.
Routine Name	Name of the given routine.
Source File	Name of the source file for the routine.
Source Line	The source files line number where the routine's implementation begins.
Token	The routine's CLR token.
Unit Name	Name of the compiled linkage unit.

Classes Category

If the *Classes* category is selected, the **Report** panel displays results for objects and the **Details** and **Call Tree** panels hold additional information about class interlinks for the class that is currently selected in the Report panel. Both the Details and Call Tree panels contain the Class Callers and Class Callees panes. The **Class Callers** pane lists all classes whose methods call methods of the given class. The **Class Callees** pane shows classes whose methods are called by methods of the given class. This information is shown in grids that hold the following columns (both Call Tree and Details have the same set of columns):

Columns (in alphabetical order)	**Description**
Call Count	The number of times methods of the given class call methods of the class selected in Report or are called by methods of that class.
Class Name	Name of the class.
Module Name	Name of the executable module where the class is defined.
Namespace	Name of the namespace to which the class belongs.

Additionally, the Details panel includes the Caller Routines, Callee Routines and Routines panes. The **Caller Routines** pane lists all methods of the class selected in the Class Callers pane. Correspondingly, the **Callee Routines** pane displays all methods of the class selected in the Class Callees pane. The **Routines** pane displays all methods of the class that is selected in the Report panel. All the three panes hold the same set of columns:

Columns (in alphabetical order)	**Description**
Call Count	*For the Routines pane:* The number of times the given method is called in the source. *For the Caller Routines pane:* The number of times the given method calls methods of the class currently selected in the Report panel. *For the Callee Routines pane:* The number of times the given method is called by methods of the class currently selected in the Report panel.

Class Name	Name of the class to which the given method belongs.
Code Type	Type of the method's code. Possible values are *x86*, *MSIL* or *PInvoke*.
Module Name	Name of the executable module where the method is defined.
Namespace	Name of the namespace to which the method's class belongs.
Routine Name	Name of the given method.
Source File	Name of the source file for the method.
Source Line	The source files line number where the method's implementation begins.
Token	The method's CLR token.
Unit Name	Name of the compiled linkage unit.

Static Analysis Profiler Options

The **Static Analysis** profiler includes options that affect the current result display. When you change these options, AQtime refreshes the data in its panels. To modify these options, use items of the **Profiler** toolbar (by default, it is displayed at the top of the **Report** panel). If this toolbar is hidden, right-click somewhere in the toolbar area and select **Profiler** from the subsequent popup list. The toolbar holds the following items (options):

- **Show addresses as RVA** – This option specifies what format the profiler should use to display addresses in the **Address** column of the Report panel. The address of each routine (or unit) consists of two components: The base address of the module, which is the address where the module is loaded in memory, and the offset of the routine relative to this base address. The offset is also called a *relative virtual address* (RVA). If *Show Addresses as RVA* is enabled, the Address column displays only relative virtual addresses. Otherwise, it displays the full routine addresses, i.e. the base address + offset. Note that the base address can be determined only after the module has been loaded into memory. Since Static Analysis does not run the executable, it uses the preferred loading address as the base one. The preferred loading address is specified in the header of the executable. To find it in AQtime, check the **Optional header** section on the **PE Information** page of the **PE Reader** panel.
- **Show routines with class names** – If it is enabled, the **Routine Name** column of the **Report**, **Details** and **Call Tree** panels for the Static Analysis profiler displays the name of the given routine along with the name of the class the routine belongs to. Otherwise, this column only displays the routine name.
- **File names with path** – If this option is enabled, the **Source File** and **Module Name** columns of the Report, Details and Call Tree panels for the Static Analysis profiler hold the entire path to the given file. Else, these columns hold the file name only.

Unused VCL Units Profiler

Unused VCL Units Profiler - Overview

This topic provides an overview of the Unused VCL Units profiler and describes the profiling results that the profiler generates. A complete profiler description includes the following topics:

- *Overview (this topic)*
- *Recognition Issues (this topic)*

- *Analyzing Profiling Results (this topic)*
- *Description of the Report Panel Columns*
- *Description of the Details Panel Columns*
- *Unused VCL Units Profiler Options*

Overview

The **Unused VCL Units** profiler helps you determine which VCL units are actually not used in your application and decrease the size of the executable (or library) by excluding those units. The Unused VCL Units profiler is a static profiler, it analyzes the application's source code and does not require the application to be running.

When a unit name is listed in the `uses` clause, the corresponding VCL unit is automatically included in the executable / library file. However, the linker does not ascertain whether the program actually calls any of the procedures from the included unit. This can occur if you drop a component onto a form to take a look at it and then you delete the component from the form. The Delphi IDE does not remove the component's unit from the uses clause.

The problem is that the Delphi compiler generates the `initialization` and `finalization` procedures for every included unit. Besides user-defined initialization / finalization code, the compiler adds its own code to manage reference count variables. Therefore, the `initialization` and `finalization` procedures are generated even if the unit does not have corresponding sections. Thus, if a unit is not actually used, then only its initialization and finalization sections are called.

The Unused VCL Units profiler includes a database that contains information about the number of procedures used by initialization and finalization sections of each standard VCL unit. This profiler compares the number of procedures exported by a unit with the number specified for this unit in the database. If the unit exports more functions than the database indicates, the profiler regards it as a used unit. If the number of the exported procedures equals the number specified in the database, the unit is included in the list of unused ones.

The analysis procedure scans the source code (namely, the `uses` section) of each user unit and of each standard unit referred by the application. Therefore, in order to use the profiler, you should specify the paths to the standard IDE units, otherwise, the profiling results will not be precise. The location where the unit sources will be sought for is defined in the Project Search Directories and Search Directories dialogs. To read the library paths specified for the compiler in the registry, you can use the dialog's **Get Defaults** button.

The number of procedures in the initialization section varies from one Delphi version to another. The profiler supports Borland Delphi versions from Delphi 3 to Delphi 2010, as well as CodeGear Delphi 2007. To specify which Delphi version was used to build the profiled application, use the *Delphi version* option.

Recognition Issues

The described analysis algorithm has one drawback: If a unit is used in the application, but does not export more functions than required for its initialization and finalization, the profiler still reports it as "unused".

This situation is possible in the following cases:

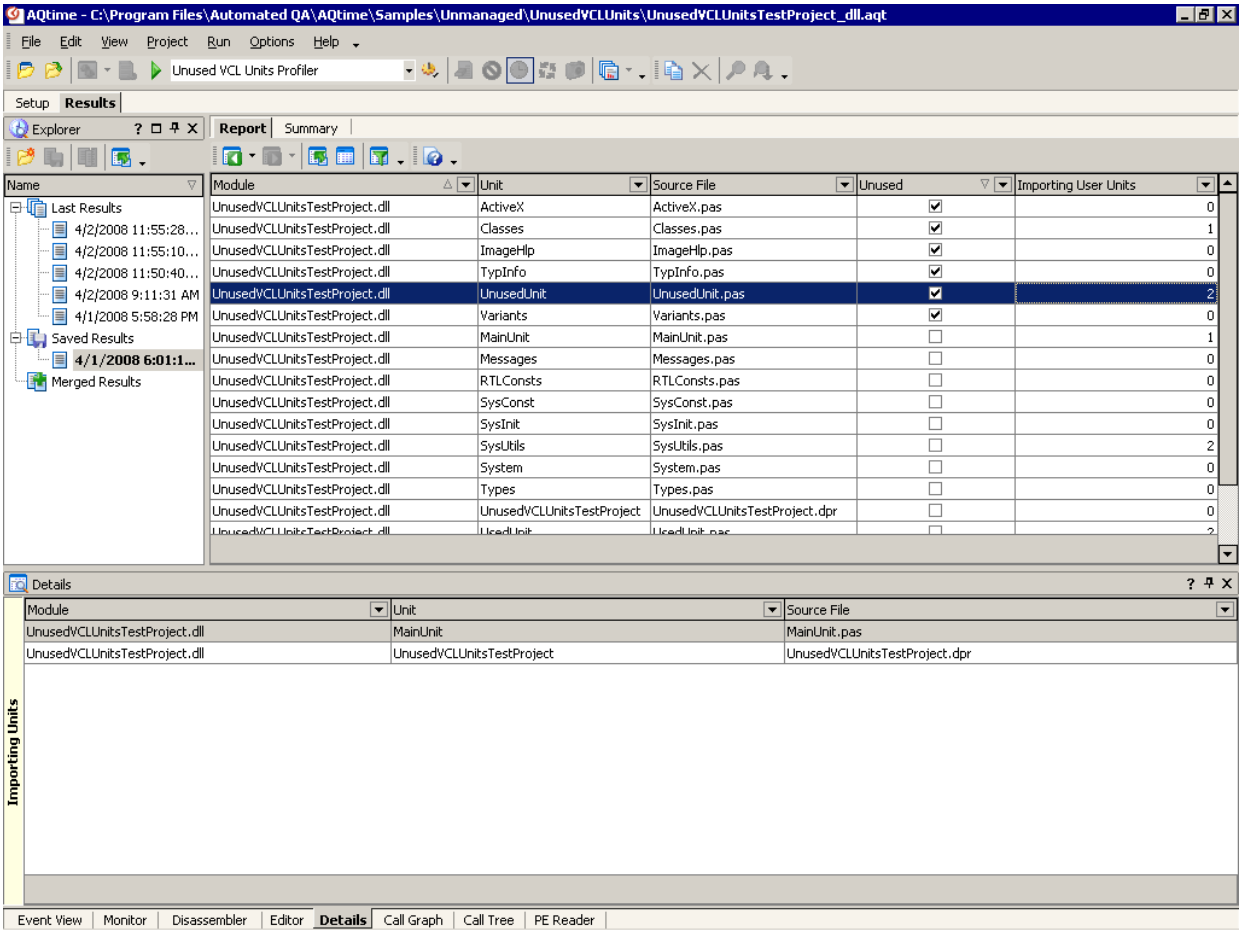
- If a unit holds constants and variables.
- If a unit declares class types that are used in other units.
- If a unit declares one or several classes that only contain inherited methods and do not define their own methods.

To resolve the issue, you can exclude these units from the analysis. This can be done by specifying unit names either in the *Ignore units with names containing* option, or in the *IgnoredUnits* text file. The *Ignore units with names containing* option defines a semicolon-separated list of words that can contain the names of units to

be skipped. Thus, you can exclude not only individual units, but also groups of units whose names match the specified pattern. For instance, by specifying *const*, you can skip both `MyConstants` and `SrvConst` units. The *IgnoredUnits.txt* file is located in the *<AQtime>\Bin\Extensions* folder and defines the names of the units to be skipped for a particular version of the compiler. All the units whose names match the value of the *Ignore units with names containing* option, or whose names are listed in the *IgnoredUnits.txt* file are considered as used.

Analyzing Profiling Results

The profiling results are displayed in the Report and Details panels. Here is a sample output of the Unused VCL Units profiler:



As you can see, the Report panel lists all of the units imported by the application and marks those units that AQtime assumes to be unused. The number of user units that refer to the chosen unit is displayed in the **Importing User Units** column. If the columns' values are zero, then the selected unit is imported indirectly (by another standard unit), otherwise, the unit name appears in the `uses` section of your application sources.

A complete list of units (both user units and standard ones) that refer to the unit selected in the Report panel is displayed in the Details panel. When a unit is recognized as unused, you can remove references to it from all user units shown in the panel.

To explore the unit's source code, double-lick its name in the Report or Details panels and switch to the Editor panel.

Keep in mind that units import one another, therefore, to remove all unused units, you should verify your application several times. The general sequence of iterations is as follows:

1. Profile the application with the Unused VCL Units profiler.
2. Open the application in the IDE and remove the previously found unused modules.
3. Recompile the application.
4. Profile the application with the Unused VCL Units profiler once again.
5. If more unused modules are found, repeat the actions starting from step 2.

For example, let's assume your project contains two unused units – `Buttons` and `Graphics` (standard VCL units). The `Buttons` unit uses some of the procedures from the `Graphics` one. So, the number of exported procedures for `Graphics` is greater than the number specified for this unit in the database. Therefore, the Unused VCL Units profiler considers the `Graphics` unit as used. After the first launch, the profiler will report that only the `Buttons` unit is unused. Remove this unit from your project, recompile your application and return to AQtime. After the second launch, the profiler will report that the `Graphics` unit is not used.

Unused VCL Units Profiler - Report Panel Columns

When displaying results generated by the Unused VCL Units profiler, the Report panel holds the following columns:

Columns (in alphabetical order)	**Description**
Importing User Units	The number of user units that refer to the current unit. The number is valid when all of the sources' paths are specified in the Project Search Directories or Search Directories dialog.
Module	The name of the application module (.exe or .dll) that contains the profiled unit.
Source File	The name of the source file where the unit is declared.
Unit	The name of the VCL unit.
Unused	Indicates whether the selected unit is used in the application. In certain cases, the profiler can report a used unit as unused. Read the description of these misleading situations in the *Recognition Issues* section of the *Unused VCL Units - Overview*.

Unused VCL Units Profiler - Details Panel Columns

When displaying results of the Unused VCL Units profiler, the Report panel lists all VCL units that your application includes. The Details panel displays a list of units that import the unit selected in the Report panel, that is, those units that contain the name of the selected unit in their `uses` section. When the selected unit is considered as unused, you can remove it from the application by excluding its name from the source code of all non-standard units listed in the Details panel. After you remove it, you should recompile your application. The list of importing units includes the following columns:

Columns (in alphabetical order)	**Description**
Module	The name of the application module (.exe or .dll) that contains the profiled unit.
Source File	The name of the source file where the unit is declared.
Unit	The name of the VCL unit.

Note: If no paths to the sources of standard units were specified in the Project Search Directories or Search Directories dialog, the profiler lacks information on how the standard units import one another, and for most of them, the Details panel displays an empty list of importing units.

Unused VCL Units Profiler Options

The Unused VCL Units profiler includes a number of options that can be customized. To modify them:

- *AQtime stand-alone*
  - Select **Options | Options** from AQtime's main menu (this will call the Options dialog),
  - Choose **Profilers | Static Analysis | Unused VCL Units** from the tree view on the left of the dialog.

  You can also modify these options by pressing **Configure Current Profiler** on the **Standard** toolbar when the Unused VCL Units profiler is selected.
- *AQtime integrated into Visual Studio*
  - Select **Tools | Options** from Visual Studio's main menu (this will call the Options dialog).
  - Select **AQtime | Profilers | Static Analysis | Unused VCL Units** from the tree view on the left of the dialog.
- *AQtime integrated into Borland Developer Studio*
  - Select **Profile | Options** from Borland Developer Studio's main menu (this will call the Options dialog).
  - Choose **Profilers | Static Analysis | Unused VCL Units** from the tree view on the left of the dialog.

In the dialog, the following options are available:

- **Delphi version** - Specifies the version of the IDE with which the profiled application was created. The profiler requires the IDE version to analyze the unit usage correctly. The possible values are: *Delphi 3, Delphi 4, Delphi 5, Delphi 6, Delphi 7, Delphi 2005, Delphi 2006, Delphi 2007, Delphi 2009, Delphi 2010*.
- **Ignore units with names containing** - Specifies the string used to exclude units from analysis. If a unit name includes a string specified by this option, the profiler considers this unit as used. You can specify several strings here. Use semicolons to separate them. The default string for the option is *const;type;messages;comstr;_TLB;* That is, the Unused VCL Units profiler ignores all the units whose names include either a *const*, *type*, *messages*, *comstr* or a *_TLB* string. An alternative to this option is specifying the names of files to be ignored in the *<AQtime>\Bin\Extensions\IgnoredUnits.txt* file.

  To learn why certain types of units should be ignored by the profiler, see the *Recognition Issues* section in the *Unused VCL Units Profiler - Overview*.

Counters Overview

This topic an overview of AQtime's counters. It includes the following sections:

- *General Information*
- *Counter Descriptions*

Counter Limitations

General Information

The Performance and Function Trace profilers can gather different kind of information about the application. What characteristic the profiler will measure depends on the selected counter. To select a counter use the *Active Counter* profiler option.

All counters work for managed and unmanaged code and support 32bit and 64bit applications. The following counters are available in the current AQtime version:

- *Elapsed Time*
- *User Time*
- *User+Kernel Time*
- *CPU Mispredicted Branches*
- *CPU Cache Misses*
- *Context Switches*
- *64K Aliasing Conflicts*
- *Split Load Replays*
- *Split Store Replays*
- *Blocked Store Forwards Replays*
- *Soft Memory Page Faults*
- *Hard Memory Page Faults*
- *All Memory Page Faults*

With the help of counters you can not only locate the application routines that are performing poorly, but investigate the reason for this performance issue. For instance, if a function operates slowly, it can be caused by inefficient code, poor memory management or a call to a slow system function. Using several different counters to profile a function, you can find out the exact reason of the delay. Advice on using counters is given in the *Searching for Bottleneck Reasons With the Performance Profiler* topic.

Counter Descriptions

- **Elapsed Time**. When you select this counter, the profiler measures the function execution time. The resultant execution time is the time span between two points in time: the entrance and the exit from the routine. This time period includes the time spent executing code in user mode, time spent executing code in kernel mode, time spent executing code in other applications, time spent switching between threads, etc. Use this counter to determine how fast your application executes the required actions in real time.
- **User Time**. This counter is also used to time the function execution. It lets you determine the “pure”execution time of your code. That is, the resultant time includes only the time spent executing code in user mode. It does not include time spent executing code in kernel mode as well as times spent executing other applications or switching between threads. The launch of several applications during profiling will not affect this counter, since it ignores the time spent executing other threads and operating system code.

  Though the *User Time* counter times the code execution in user mode only, you will see slight inconsistency in profiling results. This happens because the profiled application depends on other processes running in the system. For example, when the CPU switches the context from one thread to another, it may need to update its caches. The time spent for cache update is added to the execution time of your code.
- **User+Kernel Time**. This counter is similar to *User Time*. However, profiling results will include the time spent executing your application code as well as the time spent executing the kernel code that was called from your code. The results do not include time spent executing other applications, time spent switching between threads, etc.

© 2010 AutomatedQA Corp.

Note: Contrary to *User Time* and *User+Kernel Time* counters, *Elapsed Time* includes time, which was spent for execution code in other threads, into the function execution time. What does this mean? The CPU executes several threads concurrently by giving each thread a short period of time for execution. When the time period given to the current thread is over, the CPU switches to another thread, executes it for the short period of time, then switches to the next thread, and so on. Since the time periods are short, the threads seem to run simultaneously. Suppose now that there are 40 threads running in the system and one of these threads is your application's thread. Imagine that the CPU executed several initial instructions of the FuncA routine in your thread, but the time period given to your thread is over and the CPU switches to one of the other threads. The CPU will return to your thread and continue executing the FuncA code after it "goes" through the other 39 threads (this is assuming that all threads have the same priority). Before FuncA finishes, the CPU may switch the thread context a hundred times. If you use the *Elapsed Time* counter, the FuncA time in the profiling results will include time spent executing other threads (this will include time spent executing threads of other applications as well as time of other threads of your application). If you use *User Time* or *User+Kernel Time*, the profiling results for `FuncA` will not include this time.

The "non-time"counters work similar to *User Time* and *User+Kernel* Time. For each application routine they perform measurements "within"the routine's thread only, but not in other threads, where the CPU switched during the routine execution.

- **CPU Cache Misses**. CPU uses the cache memory to read bytes from the main memory faster. CPU loads data in cache and then works with this data, instead of reading them from the main memory. Today CPUs have several levels of cache. The CPU reads data from the first level cache. If data is not in this cache, the CPU attempts to load data from the second-level cache to the first-level cache. If there is not any data in the second-level cache, the CPU attempts to read data from the main memory or from the caches of the other levels.

  A *cache miss* is an event that occurs when the CPU is trying to read data from the cache, but this data is not in the cache. Cache misses reduce the application performance because the CPU needs to access the next-level cache or the main memory (both of which function slower, than the cache of the upper levels). Using the *CPU Cache Misses* counter you can determine how many times the CPU had to update the second-level cache during function execution. This counter helps you find routines that implement ineffective algorithms for working with memory. The better a routine operates with data in memory, the less cache misses occur during its execution.

  We would like to note that *CPU Cache Misses* counts only those cache misses that occur in the thread where your routine executes. If during the routine execution the CPU switches context to other threads, cache misses that occur in these threads will not be added to the "routine's"cache misses (see the note above).

- **Split Load Replays** and **Split Store Replays**. The cache memory is organized as a set of "lines" (the number of bytes in each line depends on the processor model). It is possible that data loaded from the memory to the cache will be stored to several cache lines. For instance, an integer value consists of four bytes. Two of these bytes can be stored to one cache line and the other two bytes can be stored to another line. A *split load* is an event that occurs when the CPU reads data from the cache and one part of the data are located in one cache line and another part - in another line. A *split store* event is similar to *split load* but it occurs when CPU writes data to the cache. These

events result in a performance penalty since the CPU reads (or writes to) two cache lines instead of one line.

The *Split Load Replays* and *Split Store Replays* counters allow you to determine whether the performance slowdowns are caused by the split load and split store events. They count replays[1] that occur due to split loads and split stores. The lower the values in profiler results, the less split load and split store events occurred during application profiling. To decrease the number of the split load and split store events, it is recommended to use the proper data alignment (for instance, 16-byte alignment) in your application.

- **Blocked Store Forwards Replays**. Use this counter to determine whether the performance slowdowns are caused by the store-to-load forwards that were blocked. Store-to-load forwarding means that the CPU forwards the store data to the load operation that follows the store.

  Store forwarding occurs under certain conditions. If these conditions are violated, store-to-load forwarding is blocked. This typically happens in the following cases (for more information, see the *Intel processor* documentation at www.intel.com):

  - The CPU reads a small amount of data and then writes more data at the same address (for example, the CPU reads one member of a structure and then writes the whole structure to the memory).
  - The CPU stores lots of data and then loads a smaller block.
  - The CPU operates with data which is not aligned properly.

  The counter measures the number of replays that occur due to blocked store forwards.

  Normally, blocked store forwards occur during each application run. However, an excessive number of replays indicates a performance issue. To avoid blocked store forwards, follow these rules where possible:

  - A load that uses store data must have the same start point and alignment that the store data has.
  - Load data must be stored in the store data.
  - Instead of several small loads after a large store to the same region of memory, use a single large load operation and then store data to the registers where possible.
  - To obtain non-aligned data, read the smallest aligned portion of data that entirely includes the desired data and then shift or mask the data as needed.

- **64K Aliasing Conflicts**. Use this counter to determine the number of 64K aliasing conflicts that occur during application profiling. A *64K aliasing conflict* occurs when the CPU reads data from a virtual address that is more than 64K apart from the previously read address. Such reading reduces the application performance since the CPU needs to update the cache. The 64K aliasing conflicts typically occur if the application works with a lot of small memory blocks that reside in memory far from one another.

- **CPU Mispredicted Branches**. Modern pipelined processors include a branch prediction unit that predicts the results of the comparison instructions. Correct prediction helps the CPU process binary instructions faster. Wrong prediction leads to the pipeline update, which results in a time penalty. In other words, code that is more predictable is executed faster than code that is not very predictable.

[1] A *replay* is an attempt of executing a micro-operation when conditions for the correct execution of this operation are not satisfied. Replays may be caused by cache misses, store forwarding issues, etc. Normally, certain number of replays always occur during the application execution. However, a superfluous number of replays designates a performance problem.

The *CPU Mispredicted Branches* counter lets you determine how well your code can be predicted by the branch prediction unit. If small values are reported, this means your application is more predictable and therefore, faster. Higher values mean that the code is not very predictable and may need to be optimized. This does not mean you need to redesign your algorithm. This just means you can speed up your code by changing the code structure. Suppose, you have the following lines of code:

```
if (a = 0)
  c = 100;
else
  c = 200;
```

If variable `a` assumes only 0 and 1 values, you can avoid the comparison by creating an array of two elements and using `a` as the array index:

```
my_array[0] = 100;
my_array[1] = 200;
...
c = my_array[a];
```

Note: For more information on CPU cache misses, split load, split store and blocked store forwarding events, 64K aliasing conflicts and on optimization of branch prediction, see the *Intel documentation* at http://www.intel.com/.

- **Context Switches**. This counter allows you to assess how the operating system schedules threads to run on the processor. A *context switch* is when the kernel suspends one thread's execution on the processor, records its current environment ("context") and restores the newly executing thread's context. For instance, this happens when a thread with a higher priority than the running thread is ready. A low rate of context switches in a multi-processing system indicates that a program monopolizes the processor and does not allow much processor time for the other threads. A high rate of context switches means that the processor is being shared repeatedly, which may cause considerable performance cost.
- **Hard Memory Page Faults**, **Soft Memory Page Faults** and **All Memory Page Faults**. If you use these counters, AQtime monitors the application execution and counts how many page faults occur. A "page fault" means that the CPU requests data from memory, but the memory page that holds this data is not available at the moment. There is a difference between "soft"page faults and "hard"page faults. A hard page fault means the operating system moved the memory page to a page file on hard disk, so to provide the requested data, it has to load the memory page from the page file. A soft page fault occurs when the desired memory page is located somewhere in memory. A soft page fault also occurs when the application allocates memory blocks. The *Hard Memory Page Faults* counter reports about hard page faults that occur during the routine execution; *Soft Memory Page Faults* - about "soft"page faults. The *All Memory Page Faults* counter is simply a sum of *Hard Memory Page Faults* and *Soft Memory Page Faults*.

  Page faults (especially hard page faults) have a dramatic impact on the application's performance. A delay that is caused by a page fault is much longer than a delay caused by a cache miss. For example, a hard page fault can take 1,000,000 times longer to process than a cache miss. Therefore, your application will be faster if there are not many page faults.

  Soft page faults occur more often than hard page faults and they are not as "dangerous". However, a lot of soft page faults can significantly slow down the application execution. Typically, a large

number of soft page faults means the application works with memory ineffectively and the algorithm of working with memory should be optimized.

Counter Limitations

There are several limitations when using counters:

- AQtime contains two packages: AQtime x86 and AQtime x64.

  If you use AQtime x86 to profile a 32-bit application on a 64-bit operating system, the only available counter is *Elapsed Time*. The other counters are not available.

  If you use AQtime x64, you can use the *Elapsed Time* counter. The other counters are only available if the 64-bit operating system is running in debug mode. For more information on profiling applications under 64-bit platforms, see *Profiling Under 64-bit Platforms*.

- AQtime supports a wide range of processors (see *Supported Processor Models*), however not all counters are available for the particular processor models.

  The **Intel Core i7**, **Intel Core 2 Duo**, **Intel Pentium II**, **Intel Pentium III**, **Intel Pentium M**, **AMD Phenom**, **AMD Athlon XP** and **AMD Athlon 64** processors support the *Elapsed Time*, *User Time*, *User+Kernel Time*, *CPU Cache Misses*, *CPU Mispredicted Branches*, *Context Switches*, *Hard Memory Page Faults*, *Soft Memory Page Faults* and *All Memory Page Faults* profiler counters, but do not support the *Split Load Replays*, *Split Store Replays*, *Blocked Store Forwards Replays* and *64K Aliasing Conflicts* counters.

  The **Mobile Intel Pentium 4**, **AMD Opteron** and **AMD Turion** processors only support the *Elapsed Time*, *Context Switches*, *Hard Memory Page Faults*, *Soft Memory Page Faults* and *All Memory Page Faults* counters.

  The **Intel Xeon** and **Intel Xeon MP** multi-core processors with the Hyper-Threading technology (for instance, Intel Xeon Duo Core) also only support the *Elapsed Time*, *Context Switches*, *Hard Memory Page Faults*, *Soft Memory Page Faults* and *All Memory Page Faults* counters. Single-core **Intel Xeon** and **Intel Xeon MP** processors support all of the counters.

  The **Intel Pentium 4** and **Intel Pentium D** processors are free from these limitations and support all profiler counters.

  If your processor supports the **SpeedStep technology**, we recommend that you turn off the dynamic CPU frequency mode before you start profiling. Otherwise, the Elapsed Time, User Time and User+Kernel Time counters may produce inaccurate timing results.

- On virtual machines you can only use the *Elapsed Time*, *Context Switches*, *Hard Memory Page Faults*, *Soft Memory Page Faults* and *All Memory Page Faults* counters. The following counters require a real CPU for timing and do not work on virtual machines: *User Time*, *User+Kernel Time*, *CPU Cache Misses*, *Split Load Replays*, *Split Store Replays*, *Blocked Store Forwards Replays*, *64K Aliasing Conflicts* and *CPU Mispredicted Branches*.

If you have Windows DDK installed, then using some counters may cause the operating system to stop unexpectedly and display the error desription on a blue screen.

To solve the problem, launch Driver Verifier (a tool from the Windows DDK package) and disable the aqIPD.sys driver verification (this driver is part of AQtime). This Driver Verifier blocks the AQtime driver.

If you cannot disable verification of the aqIPD.sys driver, you can still use the *Elapsed Time*, *Context Switches*, *Soft Memory Page Faults*, *Hard Memory Page Faults* and *All Memory Page Faults* counters.

© 2010 AutomatedQA Corp.

How To

Common Tasks

Using AQtime's profilers, you can perform the following tasks that you might need to do:

- *Disabling inlining for the managed application to be profiled*
- *Finding memory block violations*
- *Finding routines exported and imported by a module*
- *Finding the routine where an exception occurred*
- *Finding the routine that created an object or allocated a memory block*
- *Finding where a method or a class is defined in source code*
- *Knowing average, maximum and minimum execution times for a method*
- *Knowing if a method raised exceptions*
- *Knowing on which platforms your application can run*
- *Knowing paramaters and result values of function calls*
- *Knowing the number of clients that refer to an interface object*
- *Knowing the number of entries into a method*
- *Knowing the total time spent executing a method (including child methods)*
- *Knowing the total time spent on a method (excluding child methods)*
- *Knowing the structure of potential interlinks between classes*
- *Knowing the structure of references between objects*
- *Knowing the structure of routine calls in your application*
- *Knowing what binary or MSIL code a method has*
- *Knowing what libraries your application uses*
- *Knowing what methods are called the most or the least often*
- *Knowing what methods take up the most or the least execution time*
- *Knowing what methods use the most time for JIT compiling*
- *Knowing what methods were executed*
- *Knowing what source code lines are called the most or the least often*
- *Knowing what source code lines take up the most or the least execution time*
- *Knowing what source code lines were executed*
- *Searching for bottleneck reasons with the Performance profiler*
- *Searching for memory leaks*
- *Searching for resource leaks and errors in resource management functions*

© 2010 AutomatedQA Corp.

- *Tracing references between objects*

If you have not found the task you need in the list above, see other parts of the How To section.

Disabling inlining for the managed application to be profiled

The JIT (Just-In-Time) compiler can *inline* some methods when their code is short enough. The result will be that profilers can not record calls to these methods, since the calls have been replaced with code copies. The Performance and Coverage profilers includes the *Disable inlining* option that lets you avoid the problem. Checking this option will disable inlining in .NET modules profiled with the Performance or Coverage profilers. Note that enabling this option will affect performance (and thus performance measurements), because inlining normally speeds up the caller methods and also avoids certain JITting events for the inlined (called) methods.

Finding memory block violations

When an application is running under AQtime, it traces the attempts to write data beyond the allocated memory blocks. To enable the tracing, activate the *Check Memory Bounds* of the Allocation profiler. See *Checking Bounds of Memory Blocks* to learn more about this feature.

If a memory violation occurs, the Event View panel posts the following message: *"AQtime detected unexpected data written before* (or *after*) *a memory block"*, followed by the block address and it's size.

Further investigation can be made using the **Objects** category of the Report panel.

1. Select a memory violation row in the Report panel. This row holds the "*Memory Overwrite Error"* value in the class name column. For more detailed information on columns values, see the *Allocation Profiler - Report Panel Columns*.
2. Switch to the Details panel.
3. Choose the **Creation Call Stack** pane. It will display the function call's sequence for the selected violation. Since errors with the memory block bounds are only found when the corresponding memory block is deleted or reallocated the call stack displays routine calls that led to the error detection, but not to the error appearance.

Analyzing the call stack column values will help you find the exact code line where the "defective" memory block was allocated.

Finding routines exported and imported by a module

To find routines that are exported and imported by your module, use the PE Reader panel (this panel scans modules included in your AQtime project and displays information stored in those modules).

- Add your module to your AQtime project.
- Open the PE Reader panel and activate the Routine Information tabbed page. This page displays a list of imported and exported routines.
- The list of imported routines shows which functions the "parent" module imports from the selected "child" module. So, to find which functions your module imports from any other module, say *ModuleA*, select *ModuleA* in the module tree and then view the **Imported Routines** table on the Routine Information tabbed page.
- To view the function exported by your module, simply select your module in the module tree and then view the Exported Routine table on the Routine Information page.

Finding the routine where an exception occurred

When your application is being run by AQtime, the Event View panel traces all exceptions that occur in the application. To enable the tracing, activate the *Common* | *Exceptions* | *Active* panel option. If an exception occurs, the Event View panel will report about it by logging the exception code and description. The panel will also display the stack of function call that led to the exception as child nodes of the exception node (note that the call stack is traced if the panel's *Show call stack* option is enabled).

In an exception's call stack displayed in the Event View, the topmost routine is the one where the exception occurred. To view source code of a routine listed on the call stack, click this routine and then open the Editor panel (This is possible only if the application was compiled with debug information. See *How AQtime Profilers Use Metadata and Debug Information*). Debug info also lets you distinguish routines on the call stack easier. Without it, only addresses of these routines are displayed. In this instance, names are only available for functions that are exported from DLLs. With debug info, routine names are given in their natural format.

To determine the cause of the exception, examine the call stack and the conditions, in which the application was running. Note that AQtime does not support profiling of .NET applications that reside on another computer. Profiling of these applications causes an exception that occurs within the application code due to security peculiarities of the .NET Framework (see *Profiling .NET Applications - Peculiarities*).

To determine the cause of the problem, you can generate a dump file that includes information about the application's memory, threads, loaded modules and other data that may help you understand what went wrong. You can also configure AQtime so that it generates dumps automatically. See *Generating Dumps for Profiled Applications.*

Finding where a method or a class is defined in source code

For the Performance, Allocation, Coverage and Static Analysis profilers, the Report panel includes the **Module Name**, **Namespace** and **Class Name** columns, which specify the source location of each class. For the Performance, Coverage and Static Analysis profilers, the panel also includes the **Routine Name** column, which lets you locate the desired routine in profiling results.

If your application was compiled with debug information (see *How AQtime Profilers Use Metadata and Debug Information*), the Performance, Coverage and Static Analysis profilers will provide additional columns - **Source File** and **Source Line** - that specify the exact location of a particular routine in source code.

Knowing average, maximum and minimum execution times for a method

Profile your application using the Performance profiler with the *Elapsed Time*, *User Time* or *User+Kernel Time* counter. Select the *Routines* category in the Explorer panel and find the desired routine in the Report panel. Then check the **Average Time**, **Max Time** and **Min Time** columns. These are all for the method's own code, exclusive of the time spent on calling other methods. **Average Time with Children**, **Max Time with Children** and **Min Time with Children** count not only the time taken up by the method's own code, but also the entire call, including child calls.

Often the time spent on the first call to a routine might seriously differ from the time spent on subsequent calls to it, due to initializations which are normally performed during the first call. That is why it might be useful to know the time of the first call to each routine profiled. For this purpose, use the **First Time** and **First Time with Children** columns.

An alternative way to find the minimum and maximum execution time is to use the Function Trace profiler. This profiler traces the sequence of function calls and logs execution time and parameters of methods calls:

- Run the Function Trace profiler using the *Elapsed Time*, *User Time* or *User+Kernel Time* counter.

© 2010 AutomatedQA Corp.

- Generate results and select the *Call Trace* result category in the Explorer panel. The Report panel will display the sequence of function calls for the thread selected in the Explorer panel. Each row in the Report panel will correspond to a function call.
- Switch to the Report panel and filter results on the **Routine Name** column so that the panel displays rows corresponding to calls of the desired routine. See *Filtering Results* for more information.
- Sort rows on the **Time** or **Time With Children** column to find the maximum and minimum execution time for the rotuine (see *Sorting Results*).

Despite the fact that the search for minimum and maximum execution time in Function Trace results requires more operations, it still gives you one benefit: if you double-click a row in the Report panel, the Details panel will show parameter values used for the function call. That is, by analyzing the Function Trace profiler results you can find parameters passed to and received from the routine when its execution takes a maximum or minimum amount of time.

Knowing if a method raised exceptions

Use the Performance profiler. Select the *Routines* category in the Explorer panel. In the Report panel, the **Exceptions (#)** column will tell you how many exceptions were raised by each method.

Knowing on which platforms your application can run

Use the **Platform Compliance** profiler. Before running it, in the **Platform Compliance Settings** dialog, specify the platforms on which you wish to check compatibility of your application. In addition, select *Full Analysis* as the compliance level. This will perform the full analysis of all the functions that are called from statically linked libraries by your application. Once profiling is over, switch to the **Summary** panel. For each platform, this panel will give you information on functions that are not fully supported on the given platform. Therefore, you can judge whether your application can run on this or that platform.

Knowing parameters and result values of function calls

Add the desired routine to an including profiling area whose *Retrieve parameter values* property is enabled. Start the **Function Trace** profiler. Get results. Select the *Call Trace* category in the **Explorer** panel. When the Call Trace category is selected, the **Report** panel displays the sequence of function calls for each application thread. Each row in the Report panel corresponds to a function call. Select the desired thread in the Explorer panel and then choose the desired function call in the Report panel. Double-click the function call row and switch to the **Details** panel. This panel contains two tables - **Routine Parameters On Enter** and **Routine Parameters On Exit** - that display parameter values on entering and exiting the routine. The last row of the Routine Parameters On Exit table holds information on the function result value. For more information on table columns, see *Function Trace Profiler - Details Panel Columns (Call Trace Category)*.

Knowing the number of clients that refer to an interface object

AQtime includes the **Reference Count** profiler that tracks references to objects that implement the `IUnknown` interface or its descendants. Profile your application with this profiler. In the profiling results, select the *Objects* category in the Explorer panel. Then select the desired object in the Report panel and check the values of the **Total References**, **Live References** and **Peak References** columns. These values indicate the total number of references, the current number of references, and the maximum number of references that existed simultaneously.

You can trace how references to the chosen interface object were created in the **References** pane of the Details panel.

Knowing the number of entries into a method

Use the Performance or Coverage profiler. Select the *Routines* category in the Explorer panel. In the Report panel, find the line for the method and look up the **Hit Count** column.

If you want to learn the total number of potential calls to a routine as coded in the source, use the Static Analysis profiler, select the *Routines* category of profiling results, then locate the routine in the Report panel and look up the **Call Count** column.

Knowing the structure of potential interlinks between classes in your application

Use the Static Analysis profiler, select the *Classes* category of profiling results, click a class in the Report panel and then switch to the Call Tree panel. Its **Parents** pane will display the tree of classes whose methods call methods of the currently selected class as coded in the source, while the **Children** pane will display the tree of classes whose methods are called by methods of the selected class.

To know the structure of links between your classes, you can also use the **Sequence Diagram Link** profiler. This profiler statically analyzes your application to track function calls, then builds a UML diagram of function calls and outputs the diagram into Microsoft Word or Microsoft Visio. The diagram clearly shows what methods of what classes call methods of other classes.

Knowing the structure of references between objects in your application

Use the Allocation profiler. Select the *Objects* category in the Explorer panel. Then select the desired object in the Report panel and switch to the Call Tree panel:

- The **References From** table of the Call Tree panel will tell you what objects held references to the selected object, what objects referred to those objects, etc.
- The **References To** table of the Call Tree panel will tell you to what objects the selected object held references, what objects were referred to by those objects, etc.

The Allocation profiler traces references to managed objects only. If you select an unmanaged object in the Report panel, the Call Tree panel will display only the selected object, without any links to other objects.

Knowing the structure of routine calls in your application

Use the **Performance** profiler. Select the *Routines* category in the Explorer panel. Then select the desired routine in the Report panel and switch to the Call Tree panel:

- the **Parents** pane of the Call Tree panel will tell you what routines called the selected routine, what routines called those routines, etc.
- the **Children** pane of the Call Tree panel will tell you what routines were called by the selected routine, what routines were called by those routines, etc.

Note that using the Call Graph panel for the Performance profiler, you can simultaneously see several parents and children of the chosen routine, while the Call Tree panel displays the entire call hierarchy (separately for parent and child calls).

© 2010 AutomatedQA Corp.

One more way to know the structure of function calls is to run the **Function Trace** profiler. This profiler traces the sequence of function calls in your application and reports the call routes. So, run the profiler, get results, select the *Routines* category in the Explorer panel and explore all routes in the Details panel. The profiler also times each function call and logs function call parameters. This lets you learn not only the sequence of function calls, but the time taken by each call and parameters used for the call. To view all of this information, select the *Call Trace* category in the Explorer panel and then explore data in the Report and Details panels.

To learn the structure of potential (rather than actual) routine calls in your application, use the technique described above for the Performance profiler with the **Static Analysis** profiler. One more AQtime profiler, **Sequence Diagram Link**, also tracks potential calls in your application and creates a UML-style diagram showing these calls. The profiler can create diagrams in Microsoft Word or Visio, so you can select the tool that is most convenient to you.

Knowing the total time spent on a method (excluding child methods)

Profile your application using the Performance profiler with the *Elapsed Time*, *User Time* or *User+Kernel Time* counter. Select the *Routines* category in the Explorer panel. Then find the desired routine in the Report panel and check the **Time** column. To see this as a percentage of the time spent on executing all the profiled routines, check **% Time**.

To get more statistics on the methods that called the one selected in the Report panel as well as on the methods that were called by this one, refer to the Details panel.

Knowing the total time spent executing a method (including child methods)

Use the Performance profiler. Select the *Routines* category in the Explorer panel, then find the desired function in the Report panel and check the **Time with Children** column. To see this as a percentage relative to other methods, check **% with Children**. (The total against which the percentage is figured is in the footer of Time with Children. It is normally much larger than the actual runtime, since child calls are counted both for the caller and for the callee).

To get more statistics on the methods that called the one selected in the Report panel as well as on the methods that were called by this one, refer to the Details panel.

Knowing what binary or MSIL code a method has

To view the binary (or MSIL) code of a method, double-click (click) this method in the **Report**, **Setup**, **Details**, **Call Graph**, **Call Tree** or **Summary** panel and switch to the **Disassembler** panel. The panel can display the assembler instructions of the chosen routine along with its source code lines.

Knowing what libraries your application uses

Dynamic link libraries can be linked in your application both at load time and at run time.

To find libraries linked at load time, you can use the **PEReader** panel:

- Add your application to the AQtime project.
- PEReader will determine which modules the project's main module uses, which modules those modules use, etc. and build a list of used modules. To view this list, open the PE Reader panel and switch to the **Modules** tabbed page.

To find libraries linked in your application both at load time and at run time, profile your application with the **Load Library Tracer** profiler. Generate the results. The Report panel will list libraries that were loaded in memory during the application run. The panel will also report the number of loads and unloads for each library, size of the library size, preferred address and other characteristics. The Details panel will provide information on each load: the load address and the call stack.

One more way to find libraries that are linked in your application both at load time and at run time, is to profile your application with the **Performance** profiler. Whenever you run this profiler, AQtime collects information on all of the modules (managed and unmanaged) whose routines your applications calls. AQtime profiles only those routines that are selected for profiling in the Setup panel. If the **Show non-hit routines** button is released on the **Profilers** toolbar (default state for this button), the Report panel displays these profiled routines only. For other routines your application calls, AQtime does not gather information except for data that lets you identify the routine you need, as well as the class, namespace and module it belongs to. Therefore, there will be no information about time, hit count, etc for these routines. To display these routines in addition to those that are already given in the Report panel, press the Show non-hit routines button.

For .NET applications there is one more variant to learn functions of what libraries the application calls: you can make AQtime profile routines of all the managed modules that your application calls. To do this, enable *Profile Entire .NET Code* in the Setup panel. When this setting is on, all profiling areas will be ignored, though triggers will still have effect. Then use the Performance, Coverage or Allocation profiler to get the appropriate profiling results. The **Module Name** column of the Report panel will display names of all the managed modules called.

Note: Profiling with *Profile Entire .NET Code* enabled can seriously slow down your application because of the mass of information AQtime needs to collect. Once you find out which modules you need, it is recommended that you disable *Profile Entire .NET Code*, add these modules to the Setup panel and specify appropriate profiling areas.

Knowing what methods are called the most or the least often

Use the Performance, Coverage or Function Trace profiler. Select the *Routines* category in the Explorer panel. In the Report panel, sort results on the **Hit Count** column in descending order. Then check which methods are at top (most frequently called) and bottom (least frequently called). Alternatively, you can learn what routines are potentially (rather than actually) called the most or the least often as coded in the source. To do this, use the Static Analysis profiler and sort the Report panel results by the **Call Count** column.

In addition, the Summary panel displays the top 10 routines that were actually called or can be potentially called the most often. These routines are listed when you expand the node *Routines with max Hit Count* for the Performance and Coverage profilers and the node *Routines that are called most often* for Static Analysis.

If you used the Performance, Coverage or Static Analysis profiler, you can trace parent-child relations between methods and learn what methods called the one chosen in the Report panel (“parents”), and what methods were called by this one (“children”). For this, use the data shown in the Details panel. For example, for the Performance profiler the **Children** pane of the Details panel says how often the given child method was called from the selected one (the **Hit Count** column).

An alernative way to trace parent-child relationships is to analyze the Function Trace profiler results. This profiler traces the sequence of function calls and displays the call routes in results (see *Function Trace Profiler Overview*).

Knowing what methods take up the most or the least execution time

Profile your application using the Performance profiler with the *Elapsed Time*, *User Time* or *User+Kernel Time* counter. Get results. Select the *Routines* category in the Explorer panel. From the Report panel you then

have a choice of methods, depending on whether you are interested in total call time (entry to exit) or in the time taken up by the method's own code, exclusive of child calls. For total call time, sort on the **Time with Children** column. Descending will put the most expensive methods at the top, ascending will put the least expensive at the top. For own-code time, sort on the **Time** column.

Both these questions regard the *total* time cost of a method in the application, which depends more on how often the method is called, then on how slow it runs. You can optimize by improving the method code, or by checking if all those calls are necessary.

If you are interested in the individual time cost of *each* call, use the **Average Time with Children** or **Average Time** columns. But remember that optimizing a slow but seldom-called method will have negligible effect on application runtime. It may affect the user-perceived reaction time, however.

In addition, the Summary panel displays the top 10 routines whose execution time was maximal during the profiler run. These routines are listed when you expand the *Worst performance (body only)* or *Worst performance (with children)* nodes.

Often the time spent on the first call to a routine might seriously differ from the time spent on subsequent calls to it, due to initializations which are normally performed during the first call. That is why it might be useful to know the time of the first call to each routine profiled. To do this, sort profiling results on the **First Time** or **First Time with Children** column.

To avoid sorting the results of the Performance profiler, use the **More than 3% (body only)** or **More than 3% (with children)** result views. They filter the results to display only those functions that take the most time to execute their own code or their own code along with the code of all other functions they call. You can select any of these views from the **Result Views** dropdown list on the **Standard** toolbar or from the **View | Result Views** menu (if you use AQtime integrated into Visual Studio, you can select a view from the Result Views dialog that is called upon selecting the **Profile | Result Views** item of Visual Studio's main menu; if you use AQtime integrated into Borland Developer Studio, you can select a view from the Result Views dialog that is called upon clicking **Result Views** on Borland Developer Studio's **View.AQtime** toolbar). See *Result Views*.

Knowing what methods use the most time for JIT compiling

Use the Performance profiler with the *Elapsed Time*, *User Time* or *User+Kernel Time* counter. Before profiling, enable the Profile .NET runtime option. Get results and then in the Report panel, locate a record for the <JIT Compiler> fictitious routine. Switch to the Details panel and sort the records of the **Parents** pane on the desired timing column, **Time** or **Time with Children**. (The **Children** pane for <JIT Compiler> will display the methods that were called while the JIT compiler worked.) The top of the list in the Parents pane holds the most time consuming methods. You might think of preJITing them to save on run time.

In addition, you can figure out the number of times a method was JITted during the run. To do this, select the method in the Report panel, switch to Details and count how many times <JIT Compiler> is reported in the **Children** pane.

Knowing what methods were executed

Use the Performance profiler. If the **Show non-hit routines** button is released on the **Profiler** toolbar (default), then among all of the routines included in profiling, the Report panel displays only those that were executed during the run (Hit Count > 0). If this button is pressed, the Report panel will display all functions that have been executed (i.e. JIT-compiled). Note that it will even show those functions that were not included in profiling tasks. However, there will not be any information about Time or Hit Count regarding these functions.

Another way is to use the Coverage profiler: In its results all the executed methods are marked with green dots in the **Mark** column, while unexecuted ones go with red dots. (If a routine was profiled at Line Level then green dots indicate that all the method lines were executed, red dots - no line was executed and yellow dots

means that some lines were executed and some were not.) The **Hit Count** column says the same about methods: 1 - executed, 0 - non-executed.

One more way to know which methods were executed is to use the Function Trace profiler. This profiler traces the sequence of function calls and reports call routes. To view the call routes:

- Run the profiler, get results.
- Select the Routines category in the Explorer panel.
- Find the desired routine in the Report panel. The Details panel will show the call routes data.

Note, the Function Trace profiler also can trace and report the actual sequence of function calls for a thread, execution time of each function call and the call parameters. To view this information, select the *Call Trace* category in the Explorer panel and examine the data shown in the Report and Details panels.

Knowing what source code lines were executed

Make sure your application was compiled with debug information. (See *How AQtime Profilers Use Metadata and Debug Information*). Add the routines whose source code lines you want to profile to a line-level area and check this area to include it in profiling tasks. Use the Performance or Coverage profiler. Get results. Select the *Routines* category in the Explorer panel. For the Coverage profiler, the **Lines Covered**, **Lines Uncovered** and **%Covered** columns of the Report panel display the number of executed lines, the number of non-executed lines and the percentage of lines executed in each profiled routine. Click the desired routine in the Report panel and switch to the **Lines** pane of the Details panel. The **Hit Count** (for Performance and Coverage) and **Mark** (for Coverage) columns on this pane let you know if this or that line was executed (green dots, Hit Count > 1) or not (red dots, Hit Count = 0).

It may be more convenient to explore line profiling results directly in the source code: switch to the Editor panel and in the gutter you will see the hit count value or the same green and red dots next to each line of source code (Note that the Hit Count column may be not visible in the Editor's grid. To display it there, right-click within the grid, select **Field Chooser** from the context menu and then drag the column to the grid from the ensuing Field Chooser window).

Knowing what source code lines are called the most or the least often

Make sure your application was compiled with debug information. (See How AQtime Profilers Use Metadata and Debug Information). Add the routines whose source code lines you want to profile to line-level areas and check these areas to include them in profiling. Use the Performance or Coverage profiler. Get the appropriate results. Select the *Routines* category in the Explorer panel. Click the desired routine in the Report panel and switch to the **Lines** pane of the Details panel. The **Hit Count** column on this pane tells you how many times each source line was executed. You can sort the results on this pane by the Hit Count column in descending order. The most frequently executed lines will be at the top of the pane, the least frequently executed lines - at the bottom.

It may be more convenient to browse the profiling results together with source code: Move to the Editor panel and in the Editor’s grid you will see the hit count value for each source line. (Note that the Hit Count column may be not visible in the Editor’s grid. To display it there, right-click within the grid, select **Field Chooser** from the context menu and then drag the column to the grid from the ensuing Field Chooser window).

Knowing what source code lines take up the most or the least execution time

Make sure your application was compiled with debug information. (See *How AQtime Profilers Use Metadata and Debug Information*). Add the routines whose source code lines you want to profile to a line-level

area and check this area to include it in profiling. Use the Performance profiler with the *Elapsed Time*, *User Time* or *User+Kernel Time* counter. Get results. Select the *Routines* category in the Explorer panel. Click the desired routine in the Report panel and switch to the **Lines** pane of the Details panel. If you are interested in the total time of line execution, sort results on this pane by the **Time** column. If you are interested in the total time spent on executing the line and all the routines it called, sort the results by the **Time with Children** column. If you sort in descending order, the slowest lines will be at the top of the pane, the fastest - at the bottom.

It may be more convenient to browse line profiling results together with source code: Switch to the Editor panel and in the Editor's grid you will see the profiling results next to each source line. (Note that by default not all columns are visible in the Editor's grid. To display a column there, right-click within the grid, select **Field Chooser** from the context menu and then drag the column to the grid from the ensuing Field Chooser window).

Searching for memory leaks

To trace the memory usage in your application, run AQtime's **Allocation** profiler. This profiler monitors the application execution and tracks the allocations and deallocations of memory blocks as well as the creations and deletions of objects in both managed and unmanaged code. For more information about the memory management allocations that the profiler traces, see the *Allocation profiler description* and *Checking Bounds of Memory Blocks*.

Note that the Allocation profiler will trace the creation and deletion of those objects, which class names are added to profiling areas of the class type in the **Setup** panel. Therefore, before you run the profiler, please add the desired classes to class profiling areas (see *Defining Areas to Profile*). Tracing the allocations of memory blocks does not require any preparations to be made in the Setup panel.

When you run the Allocation profiler, AQtime can generate results both during the profiler run and after the application terminates. Results that were obtained during the run allow you see what objects currently exist in memory. Results that were obtained after the application termination help you find memory leaks.

Profiling results are organized into two categories: *Classes* and *Routines*. You can select the desired category in the **Explorer** panel.

When the *Classes* category is selected, the Report panel lists all classes whose instances were created during the application run. In addition, the panel displays information about calls to memory management routines that allocated memory blocks during the run. The class name is displayed in the **Class Name** column (for memory blocks this column holds either *C++ native memory*, *VCL native memory*, or *VB native memory* value depending on what functions allocated memory blocks). The **Live Count** column shows the number of object instances and memory blocks that currently exist in memory. That is, the value of this column is greater than zero, if the given class has instances that were not destroyed by the moment of results generation. You can filter or sort results on the Live Count column to quickly find classes, which instances were not destroyed. The **Live Size** column informs you about the amount of memory occupied by these class instances and memory blocks.

To view the Live Count and Live Size information in real-time, during the profiling process, use the **Monitor** panel. See *Using the Monitor Panel With the Allocation Profiler*.

When the *Objects* category of profiling results is selected, the Report panel lists memory blocks and class instances (i.e. objects) that were allocated (created) and not destroyed during the application run. Identifiers of objects and memory blocks are displayed in the **Object Name** column. They have the form *class_name.nn*. For instance, the name *String.5* means the fifth `String` object created after profiling started. For memory blocks this column holds values like *C++ native memory.4* or *VCL native memory.10*. These mean the 4th memory block allocated with a C++ operator (e.g. `new`) or 10th memory block allocated with a VCL memory management routine (e.g. `GetMem`).

To find which routine allocated a memory block or created an object instance:

- Select the *Objects* category in the Explorer panel.

- Click the desired block or object in the Report panel and then switch to the **Details** panel.
- Switch to the **Creation Call Stack** pane of the Details panel (note that this page is empty if the Allocation profiler's *Collect stack information* option is set to *None*).
- The Creation Call Stack pane displays the stack of function calls that led to the object creation (memory block allocation). The routine that created the object (allocated the memory block) occupies the topmost row.
- To view the source code of a routine, double-click the routine in the call stack. AQtime will bring up the **Editor** panel and position the cursor on the first line of the routine's source code (if you profile a .NET application, then you need to compile it with debug information in order the Editor can show the application source code. See *How AQtime Profilers Use Metadata and Debug Information*).

Searching for resource leaks and errors in resource management functions

To trace the resource usage in your application, run AQtime's **Resource** profiler. This profiler monitors the application's execution and tracks how the entire application (rather than any part of it) allocates and deallocates Windows resources (menus, bitmaps, pens, etc.). The profiler also lets you locate errors in resource-related functions. The profiler supports both managed and unmanaged applications.

When you run the Resource profiler, AQtime can generate results both during the profiler run and after the application terminates. Results that were obtained during the run allow you see what resource instances currently exist. Results that were obtained after the application termination help you find resource leaks.

Profiling results are organized into three categories: *Classes*, *Objects* and *Errors*. You can select the desired category in the **Explorer** panel.

When the *Classes* category is selected, the **Report** panel lists all resource types whose instances were created during the application run. The resource type name is displayed in the **Class Name** column. The **Live Count** column shows the number of resource instances that currently exist. That is, the value of this column is greater than zero, if the given resource type has instances that were not deallocated by the moment of results generation. You can filter or sort results on the Live Count column to quickly find resource types whose instances were not deallocated. The **Live Size** column informs you about the amount of memory occupied by these resource instances.

When the *Objects* category of profiling results is selected, the Report panel lists resource instances that were allocated and not deallocated during the application run. Resource instance identifiers are displayed in the **Object Name** column. They have the form *resource_type_name.nn*. For instance, the name *Icon.5* means the fifth instance of the Icon category created after the profiling started.

To find out which routine allocated a resource instance:

- Select the *Objects* category in the Explorer panel.
- Click the desired resource instance in the Report panel and then switch to the **Details** panel (note that the **Creation Call Stack** pane of this panel is empty if the Resource profiler's *Collect stack information* option is set to *None*).
- The Creation Call Stack pane displays the stack of function calls that led to the resource instance creation. The routine that created the resource instance occupies the topmost row.
- To view the source code of a routine, double-click the routine in the call stack. AQtime will bring up the **Editor** panel and position the cursor on the first line of the routine's source code (if you profile a .NET application, then you need to compile it with debug information in order the Editor

can show the application source code. See *How AQtime Profilers Use Metadata and Debug Information*).

When the *Errors* category of profiling results is selected in the Explorer panel, the Report panel lists all errors that occurred in the resource management functions which the Resource profiler traces (See *Resource Profiler - List of Checked Functions*). To learn in which routine the given error occurred, have a look at the Routine Name column. You can view the MSDN topic about this routine. To do this, click the link in the Reference column. A description of the error is shown in the **Description** column.

Tracing references between objects

The method described below is possible for managed applications only. Use the Allocation profiler to get appropriate results. When the *Classes* category of profiling results is selected, the Report panel lists all classes whose instances were created during the application run. If the given class has instances that were not destroyed, the value of the **Live Size** column is greater than zero. When the *Objects* category of profiling results is selected, the Report panel lists class instances (i.e. objects) that were created and not destroyed during the application run. Object identifiers are displayed in the **Object Name** column. For instance, the name *String.5* means the fifth `String` object created after profiling started. The **References To** and **Referenced By** columns specify the number of objects to which the selected object refers and the number of objects that refer to the object itself.

Click an object and switch to the Details panel. Each of the **References To** and **References From** panes holds a list of objects. The first list is for objects that are referred by the selected object. The second list is for objects that refer to the selected object. You can also switch to the Call Graph or Call Tree panel to explore references between objects.

Profiling Applications

Optimizing the Profiling Process

Below you will find some tips for getting the most out of a profile cycle in AQtime, with the least amount of wasted effort:

- Do not assume a function has “no problems”. The profilers are there to give you a health report; use them. Known problems may have unexpected roots.
- Even a small, quick function can impair performance if it is called extensively. Always check the Hit Count for anomalies. Are there errant hit counts, for instance where a very common piece of code makes a needless call? If they are not errant, can the very high hit counts be decreased by tuning your algorithms?
- Restrict you profiling areas when you can. The profilers need time to gather information about the sections they are set to analyze (areas). They may take more time while profiling the actual execution. The more precise your area specification, the faster the profiling. Remember that an application includes a lot of code that will never need to be profiled. For instance, user interface code normally does little but wait on the user. See *Controlling What to Profile*.
- Long functions can be difficult to profile. One way around this is to break them down into several sub-functions for profiling purposes (You will probably find that the code is also clearer once broken down, and easier to analyze).
- No real-world execution of an application is identical to another. For instance, it’s impossible to predict and control how many times the CPU updates its caches. So that from one run to the next,

you should expect some inconsistency in results. Where you need high-precision results, keep a very detailed record of how the test was run.

- To help achieve consistent test results, reduce the number of processes running on your machine during profiling.

Getting Results During Testing

AQtime normally generates results once the profiled application has ended its run. When profiling a dynamic link library, this means results are generated when the host application exits.

However, you might be profiling an application that does not stop until the system shuts down (for example, an NT service), or you may wish to obtain results without closing the running application (for instance, the host for a dll).

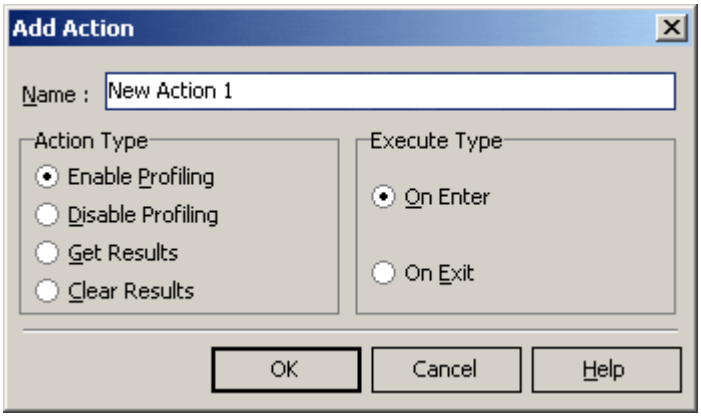
You can command AQtime to generate results by using the Get Results menu item, by using specific actions, or by controlling AQtime from your application. This topic describes all of these ways and also provides a brief overview of the Clear Results feature (see *Clearing Results During Profiling*).

Get Results Menu Item

To command AQtime to generate profiling results, select the **Run | Get Results** item from AQtime's main menu or press **Get Results** on the Standard toolbar. If you use AQtime integrated into Visual Studio, you can do this by selecting **Profile | Get Results** option from Visual Studio's main menu; if you use AQtime integrated into Borland Developer Studio, you can do this by clicking **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar. Results will be generated as if the profiled application had terminated.

Using Specific Actions

By using AQtime actions you can configure the profiler run so that AQtime will perform the following specific operations upon entering or exiting a routine:



So, you can create an action of the *Get results* type and add the desired routine or routines to this action. The routine(s) that you select for the action must be executed during the profiler run. Depending on the *Execution Type* property of the action, AQtime will generate results upon entering or existing the routine(s) added to the action.

For more information on actions, see *Using Actions*.

Commanding AQtime to Generate Results via COM

AQtime is a COM server. You may connect to it via COM and use the COM interfaces command to command the profiling engine to generate results. The whole procedure includes the following steps:

- Connect to AQtime via COM (you can use the *AQtime.AQtime* program identifier).
- Obtain the `IntegrationManager` object.
- To generate results, call the `TakeSnapshot` method of the `IntegrationManager` object.

For more information on the `IntegrationManager` object and the `TakeSnapshot` method, see *Working With AQtime via COM*.

Commanding AQtime to Generate Results from a Tested Application

You may also command AQtime to generate results from your profiled application. To do this:

- Open your application's project in the development tool you use.
- Include the *AQtimeHelpers* file into your application project. This file is located in the following folder:

If you use...	Add the following file
Microsoft Visual C# .NET	<AQtime>\SDK\CS\AQtimeHelpers.cs
Microsoft Visual Basic .NET	<AQtime>\SDK\VBNET\AQtimeHelpers.vb
Microsoft Visual C++, Borland C++Builder, Borland C++ or Intel C++	<AQtime>\SDK\CPP\Common\AQtimeHelpers.cpp
Microsoft Visual Basic	<AQtime>\SDK\VB\AQtimeHelpers.bas
Borland Delphi	<AQtime>\SDK\Delphi\Common\AQtimeHelpers.pas

  The file contains the declaration of the **`GenerateResults`** function. This function commands AQtime to generate and display profiling results that have been accumulated by the call. The function does not use any parameters.

- Call the `GenerateResults` function in your code.

  If your application already contains a routine with the name `GenerateResults`, you may need to use the `AQtimeHelpers` namespace or the `AQtimeHelpers` unit name (this depends on your compiler) when calling the `GenerateResults` routine:

```
[Visual C++]
. . .
// Call GenerateResults using the namespace
AQtimeHelpers::GenerateResults();
```

```
[Delphi]
 . . .
// Call GenerateResults using the unit name
AQtimeHelpers.GenerateResults();
```

About Clearing the Results

Note that if you do not want to keep the previous results, you can flush them before getting the most recent results. This can be done by selecting the **Run | Clear Results** item from AQtime's main menu or by pressing the **Clear Results** button on the Standard toolbar (if you use AQtime integrated in Visual Studio, select the

Profile | Clear Results item from Visual Studio's main menu or click **Clear Results** on the AQtime toolbar. If you use AQtime integrated into Borland Developer Studio, click **Clear Results** on Borland Developer Studio's **Run.AQtime** toolbar). See *Clearing Results During Profiling* for more information on this.

Clearing Results During Profiling

The profiling engine accumulates results during profiling and displays them in the {Report}, {Details} and other panels when the profiled application is over or when AQtime gets a command to generate the results. AQtime provides a way to clear the accumulated results. This feature can be useful if you want AQtime to clear results accumulated by some point in time and display only the results that we collected after this point.

You can command AQtime to clear results by selecting the **Run | Clear Results** item from AQtime's main menu or by pressing the **Clear Results** button on the Standard toolbar (if you use AQtime integrated into Visual Studio, you can do this by selecting **Profile | Clear Results** option from Visual Studio's main menu; if you use AQtime integrated into Borland Developer Studio, you can do this by clicking **Clear Results** on Borland Developer Studio's **Run.AQtime** toolbar).

Another way to clear results when profiling is to create an action that will command AQtime to flush the gathered results (Action Type: Clear Results). The routine that you select for this must be executed during the profiler run. For more information on actions, see *Using Actions*.

Note that if you run the Allocation profiler and the profiler's *Check memory bounds* option is enabled, the results cannot be cleared.

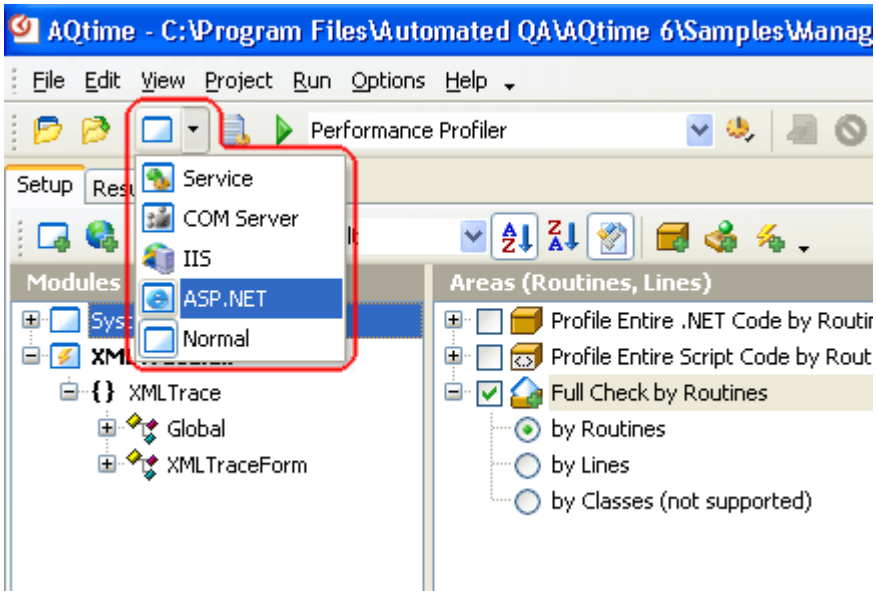
Profiling Modes

AQtime can work in any of the following modes (the appropriate mode depends on the type of application you are going to profile):

- **Normal** - This is AQtime's default mode, which is used to profile ordinary applications: managed and unmanaged executables and libraries.
- **COM Server** - This mode is used to profile COM servers of any type (in-process, out-of-process, DCOM, COM+ or MTS). See *Profiling COM Applications*.
- **ASP.NET** - This mode is used to profile ASP.NET applications and .NET Web services. See *Profiling ASP.NET Applications*.
- **Service** - This mode is used to profile Windows NT services. See *Profiling Services*. Note that this mode is not intended for ASP.NET service profiling.
- **IIS** - This mode is used to profile IIS applications and Web services created with unmanaged compilers. See *Profiling IIS Applications*.

To select the desired mode, use the **Profiling Mode** list that is located on AQtime's **Standard** toolbar:

© 2010 AutomatedQA Corp.



If you are using AQtime integrated into Microsoft Visual Studio, the **Profiling Mode** list is located on Visual Studio's **AQtime** toolbar. In Embarcadero RAD Studio, the **Profiling Mode** list resides on the **Run.AQtime** toolbar.

"Attach to Process"

Once you press the **Run** button to start profiling, AQtime launches your application, instruments it (prepares it for profiling) and then starts collecting statistics on the applications execution. Though almost all applications can be started under AQtime, doing this can be a non-trivial task for some types of applications.

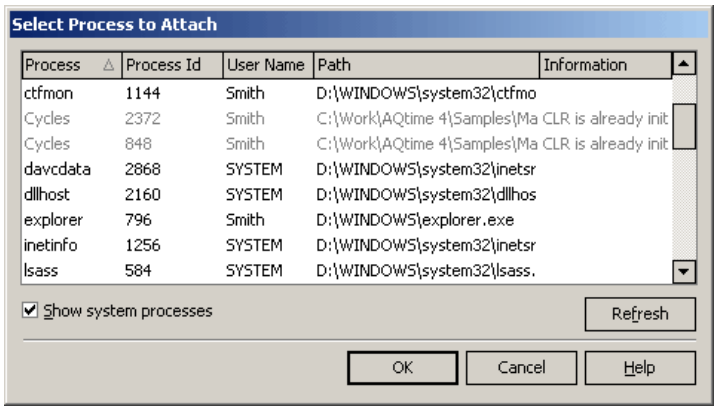
AQtime offers the "Attach to Process" feature that makes things easier. Profiling with this feature means that the profiled application is not started using AQtime: AQtime can connect to the application and instrument it on the fly. This feature is supported by the **Performance**, **Coverage**, **Exception Tracer**, **Function Trace**, **Load Library Tracer**, **Allocation** and **Resource** profilers.

Profiling with the "Attach to Process" feature does not differ much from ordinary profiling. Below is a step-by-step explanation of how to profile an executable using this feature. Two notes before we proceed:

- AQtime cannot attach to a process that is running under another debugger, for instance, a process which is running under Microsoft Visual Studio.
- If you attach to a managed process, AQtime will not analyze managed code in this process. Only unmanaged code will be profiled. Attaching to a managed process may be useful, if you need to profile a native-code dynamic link library that is used by a .NET application.

1. Compile the application with the debug information.
2. Open the application in AQtime.
3. Switch to the Setup panel and select areas and triggers for profiling (see *Controlling What to Profile*).
4. Select **Normal** from the **Profiling Mode** dropdown list box on AQtime's Standard toolbar (from the **AQtime** toolbar in Visual Studio or from the **Run.AQtime** toolbar in Borland Developer Studio). This is necessary because the Attach to Process feature only works in the Normal mode.
5. Select the desired profiler.
6. Run the application outside AQtime.

7. Select **Run | Attach to Process** from the main menu or press **Attach to Process** button on the **Standard** toolbar (select the **Profile | Attach** from Visual Studio's main menu or click **Attach to Process** on Borland Developer Studio's **Run.AQtime** toolbar). If the selected profiler does not support the "Attach to Process" feature, this item is disabled. Upon selecting this item, the following dialog will appear:



The dialog holds a list of all of the processes that exist in the operating system at the moment. By default, system processes are hidden. To make them visible, check the **Show system processes** box. For instance, if you are profiling an IIS application (IIS applications are DLLs), you should check this box, because normally the IIS process is running under the system account.

Note once again that AQtime cannot attach to a process that is running under a debugger. Such processes are grayed out in the dialog. The explanation why AQtime cannot attach to a process is shown in the **Information** column of the dialog.

If you are profiling a dynamic link library, you should attach to the process that uses it. To attach to the desired process, select it in the list and press **OK**.

If you attach to a managed process, AQtime will display a message informing you that managed code will not be profiled (see the note about this above). The reason for attaching in this case is to profile unmanaged DLL(s) loaded by that managed process.

If the desired process is not running, you should start it, then return to the dialog and press the **Refresh** button or F5 on the keyboard to update the process list. When the process is displayed in the list, you can attach to it. The **Cancel** button will cancel profiling.

8. Upon pressing **OK**, AQtime injects its DLL into the process address space and then starts instrumenting the profiled module. If the profiler's option *Show uninstrumented routines | In the dialog* is on, then AQtime displays the Uninstrumented Routines Dialog explaining why certain routines cannot be profiled. To start profiling, press **OK** in the dialog.

Now you can profile the application as your needs dictate.

The profiling starts after the instrumentation finishes.

Important: all calls to functions (lines) that occurred before instrumentation will not be profiled and will not be included in the profiling results. For instance, a trigger will not affect profiling, if it is started while this trigger is running. In addition, a routine that was called before or during the profiling start will not be shown in the **Parent** and **Children** tables of the Details panel.

Note: In 64-bit versions of Windows, 32-bit modules can be loaded into 32-bit processes only and 64-bit modules can be loaded into 64-bit processes only. So, if the "bitness" of your module does not match the "bitness" of the process, to which the module is

loaded, AQtime will not start profiling.

To obtain the profiler results, select **Run | Get Results** from AQtime's main menu (select **Profile | Get Results** from Visual Studio's menu or click **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar). An alternative way to obtain profiling results is to create an action for this. See *Getting Results During Testing* for more information. Results are also generated when the process, to which you attached, finishes.

Profiling Under 64-bit Platforms

AQtime can profile 64-bit applications, as well as COM, ASP.NET, IIS and service applications on 64-bit versions of the Windows operating systems (for information on supported versions, see *System Requirements*). Profiling of 64-bit applications has some peculiarities that are described further in this topic.

Profilers for 64-bit Applications

Currently, 64-bit applications are supported by the following profilers:

- Performance Profiler
- Allocation Profiler
- Resource Profiler
- Static Analysis Profiler
- Coverage Profiler
- Exception Trace Profiler
- Function Trace Profiler

Profiling Under Windows Server 2003

To profile 64-bit applications under Windows Server 2003 x64 edition, you should log into the session 0 (console session). To do this, you can run the Remote Console via the following command line:

```
mstsc.exe /console
```

Profiling Under Windows Server 2008 R2

In order for AQtime to be able to function on Windows Server 2008 R2, the WoW64 component of this operating system must be installed. The Server Core installation option for Windows Server 2008 R2 does not install WoW64 by default, so, you should install it yourself. This requirement concerns both AQtime x86 and x64 packages.

Using Counters

The 64-bit versions of Windows XP, Windows Server 2003 Service Pack 1 and later 64-bit versions of Windows operating systems have the Kernel Patch Protection (KPP) feature. Due to this feature counters, with the exception of the *Elapsed Time* counter, may be unstable and cause a system crash. Only the *Elapsed Time* counter works properly and does not cause any problems with KPP and we recommend that you use it for analyzing the performance of your applications. Using other counters may cause a crash on 64-bit operating systems.

If you need to use counters other than *Elapsed Time*, then a possible workaround is to run the operating system in debug mode. In this case, you will be able to use all the counters.

Note however that this approach will disable .NET debugging: the debugger of Microsoft Visual Studio will be disabled (that is, Visual Studio will not debug managed code) and AQtime's Event View panel will not trace and report .NET-specific events. Though the profiling of managed code will work.

The way you enable debug mode depends on your operating system:

Running Window 7 and Windows Vista in debug mode

To activate debug mode in Windows 7 or Windows Vista:

- Log onto Windows Vista.
- Open the Command Prompt window and type the following commands in it:

```
Bcdedit /debug ON

Bcdedit /dbgsettings SERIAL DEBUGPORT:1 BAUDRATE:115200 /start AUTOENABLE
```

- Restart the operating system.

To turn off the debug mode:

- Open the Command Prompt window and execute the follow command:

```
Bcdedit /debug OFF
```

- Restart the operating system.

Running Windows XP in debug mode

To enable debug mode in Windows XP, modify the boot.ini file. It resides in the root of the operating system's disk. For instance, if you have Window XP installed on drive *C:*, the file name is *C:\boot.ini*.

- Open boot.ini in Notepad.
- Find the `[operating systems]` section. Items of this section correspond to the items of the boot menu.
- Find the desired item and add the */debug=autoenable* parameter to it, for instance:

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP
Professional" /noexecute=optin /fastdetect /debug=autoenable
```

- Save the changes.
- Restart the operating system.

To disable the debug mode, remove the */debug=autoenable* parameter from *boot.ini*.

Profiling Dynamic Link Libraries

Using AQtime you can profile dynamic link libraries that are linked to your executable both at startup time (so called statically linked DLLs) and at run time (dynamically linked DLLs). Profiling a DLL is similar to profiling of any other standard application. In some cases it is even simpler, because the DLL may be compiled without debug information. This topic explains how you can profile dynamic link libraries with AQtime.

In further explanations we will call the executable that loads the profiled DLL in memory a *host application*. The manner in which you profile your DLL depends on whether you will start the DLL's host application from AQtime or you will attach to it at run time.

If you can start the host application from AQtime, then you can profile your DLL in the following manner:

- Compile your DLL. See *How AQtime Profilers Use Metadata and Debug Information* to decide if you need to include debug information.

- Load the DLL in AQtime as a project (see *Opening a Project*).
- Open the Run Parameters dialog and specify the host application for a dynamic link library. When you start profiling, AQtime launches the specified host application. The host application then calls the functions defined in the profiled dynamic link library. Note that AQtime does not profile the host application, so it can be compiled without the debug info.
- Continue profiling as you normally would. AQtime profiles DLL functions only when they are included in one of profiling areas or when *Full Check* is active.

  Note: In 64-bit versions of Windows, 32-bit modules can be loaded into 32-bit processes only and 64-bit modules can be loaded into 64-bit processes only. So, if the "bitness" of your dynamic link library does not match the "bitness" of the process, to which the library is loaded, AQtime will not start profiling.

AQtime normally generates results once the profiled application has ended its run. When profiling a dynamic link library, this means results are generated when the host application exits. If you need to obtain results without closing the host application, you can do this by using the **Run | Get Results** option from AQtime's main menu (select **Profile | Get Results** in Visual Studio or click **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar) or by creating an action that will command AQtime to generate results. For more information, see *Getting Results During Testing*.

If the host application is already loaded into AQtime as a project, you can include your DLL into profiling tasks by adding it to that AQtime project. To do this, choose **Add Module** from the Setup toolbar or from the context menu of the Setup panel and select the desired DLL in the subsequent Open File dialog. Now, when you start profiling in AQtime, it will launch the host application, which, in turn, will load the DLL in memory and call its functions.

Note that if you add a DLL to the host application's project, AQtime can use both DLL's debug information and table of exported functions to determine the routine's location in code. That is, the added DLL may be compiled without debug information and may still be profiled by AQtime. This makes it possible to profile dynamic link libraries, which are used by your application and which do not have debug info (for example, system libraries. See *Profiling System Calls*). However, the absence of debug information imposes some limits on the profiling:

- You can profile only those functions that are exported by the DLL. You cannot profile "internal" DLL functions and procedures.
- You can profile DLL functions at routine level only.
- The Allocation profiler will not trace memory allocations that are done directly within the DLL code.

For .NET modules AQtime offers one more way for profiling the dynamic link libraries: you may check the **Profile Entire .NET Code** box in the Setup panel. If this box is checked, AQtime will profile all managed routines and modules that are used by the profiled executable regardless of whether they are added to profiling areas or not. You can select the profiling level, at which AQtime will profile managed routines, by selecting *by Routines*, *by Lines* or *by Class* options under the *Profile Entire .NET Code* node.

One more way to profile DLLs with AQtime is to attach to the process that uses this DLL:

- Compile your DLL. See *How AQtime Profilers Use Metadata and Debug Information* to decide if you need to include debug information.
- Load the DLL in AQtime as a project (see *Opening a Project*).
- Select **Run | Attach to Process** from the main menu or press **Attach to Process** button on the **Standard** toolbar (select the **Profile | Attach** from Visual Studio's main menu or click **Attach**

to Process on Borland Developer Studio's **Run.AQtime** toolbar). This will call the **Select Process to Attach** dialog that lists all currently running processes.

- In the dialog, select the process to which you would like to attach your module and press **OK**. If the desired process is not in the list, launch it outside of AQtime, and then return to the Attach to Process dialog and press **Refresh** to update the list.
- Profile your applications you normally would.
- To obtain the profiler results, select **Run | Get Results** from AQtime's main menu (select **Profile | Get Results** if you use AQtime integrated into Visual Studio; click **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar if you use AQtime integrated into Borland Developer Studio). Results are also generated when the process, to which you attached the DLL, finishes.

Profiling Web Server Applications

Web server applications are normally dynamic link libraries that are loaded and called by an HTTP server. They are used to enhance the server capabilities. For instance, a Web server application can process user requests and dynamically generate Web pages with the appropriate content. Web server applications can be created with different compilers and can be meant for HTTP servers of different types (for Internet Information Services, Apache, etc.) For instance, using Microsoft Visual Studio, you can create an IIS application that uses the functionality provided by the .NET Framework and .NET assemblies.

The manner in which you profile your Web application with AQtime depends on the server type. For profiling Web applications that work with Internet Information Services, AQtime offers special profiling modes: IIS and ASP.NET. For detailed information on profiling these applications, see the following topics:

- *Profiling IIS Applications*
- *Profiling ASP.NET Applications*

A general recipe for profiling Web applications is rather simple: since a Web application is a DLL, you can profile it in the same manner as you profile any other DLL. To do this, you can either run the HTTP server process under AQtime or attach to that process at run time. The way you choose depends on the situation.

If you fully control the HTTP server, you can try to run it under AQtime:

- Compile your Web application with debug information.
- Load your application in AQtime.
- Select the **Normal** profiling mode.
- Open the Run Parameters dialog and specify the fully qualified name of the HTTP server executable in the **Host Application** edit box. Note that depending on their options, HTTP servers can use different process names and different executables.
- Click **Run** to start profiling (**Profile | Run** in Visual Studio or **Run | Run With Profiling** in Borland Developer Studio).
- In your Internet browser, open the Web page that loads your Web server application. Then profile your application as you usually would.

  Since the process that uses your DLL may never end by itself, you may need to use the **Run | Get Results** menu item (**Profile | Get Results** in Visual Studio or **Get Results** on the **Run.AQtime** toolbar in Borland Developer Studio) in order to obtain profiling results. Another way to obtain the results is to create an action that will "tell" AQtime to generate the results. See *Getting Results During Testing* for more

information.

If for some reason the HTTP server process cannot be launched under AQtime, you can try attaching your Web server DLL to it:

- Compile your Web application with debug information.
- Load your application in AQtime.
- Select the **Normal** profiling mode.
- Select **Run | Attach to Process** from the main menu or press **Attach to Process** button on the **Standard** toolbar (select the **Profile | Attach** from Visual Studio's main menu or click **Attach to Process** on Borland Developer Studio's **Run.AQtime** toolbar). This will call the **Select Process to Attach** dialog listing all the processes that are running at moment.
- In the dialog, select the process of the HTTP server (to be more exact, you should select the process that will load your DLL in memory) and press **OK** to attach to this process.
- In your Internet browser, open the Web page that loads your Web server application. Then profile your application as you usually would.
- To obtain profiling results, use the **Run | Get Results** menu item (**Profile | Get Results** in Visual Studio or **Get Results** on the **Run.AQtime** toolbar in Borland Developer Studio). Another way to obtain the results is to create an action that will "tell" AQtime to generate the results. See *Getting Results During Testing* for more information.

Profiling ASP.NET Applications

ASP.NET applications can be seen as Internet Information Services (IIS) applications that are built with .NET compilers and that use functionality provided by the .NET Framework and .NET assemblies. Despite the fact that ASP.NET applications are similar to IIS applications, they are profiled in a different manner. This topic describes how to profile ASP.NET applications or services with AQtime. For information on how to profile ordinary (unmanaged) IIS applications, see *Profiling IIS Applications*.

Profiling ASP.NET applications requires 32- or 64-bit editions of Windows 7, Windows Server 2008, Windows Vista, Windows XP, Windows Server 2003 or Windows 2000 with Internet Information Services ver. 4.0 - 7.0.

The manner in which you profile your application depends on the development tool that you use to create the ASP.NET application. Any ASP.NET application created in any development tool can be profiled using IIS. Additionally, Visual Studio 2005 has its own server - *ASP.NET Development Server* - that is generally used instead of IIS to develop and debug ASP.NET applications. However, it is still possible to use IIS to profile your application outside of Visual Studio 2005. The following topics provide instructions on different ways to profile ASP.NET applications.

- *Profiling IIS Applications*

  Describes how to profile ASP.NET applications or services using the ASP.NET Development Server.

- *Profiling ASP.NET Applications*

  Describes how to profile ASP.NET applications or services using the Internet Information Services (IIS).

AQtime includes a number of samples that you can use to get started profiling ASP.NET applications.

XMLTrace

This sample parses an XML file and displays its contents and structure on screen.

You can find the application sources in the following folders:

- Microsoft Visual Studio .NET 7.x projects:
  - <AQtime>\Samples\Managed\VS.NET\ASP.NET\XMLTrace\VB\ - Microsoft Visual Basic .NET
  - <AQtime>\Samples\Managed\VS.NET\ASP.NET\XMLTrace\CS\ - Microsoft Visual C# .NET
  - <AQtime>\Samples\Managed\VS.NET\ASP.NET\XMLTrace\JS\ - Microsoft Visual J# .NET
- Microsoft Visual Studio 2005 projects:
  - <AQtime>\Samples\Managed\VS2005\ASP.NET\XMLTrace\VB\ - Microsoft Visual Basic .NET
  - <AQtime>\Samples\Managed\VS2005\ASP.NET\XMLTrace\CS\ - Microsoft Visual C# .NET
  - <AQtime>\Samples\Managed\VS2005\ASP.NET\XMLTrace\JS\ - Microsoft Visual J# .NET

MasterDetails

This sample demonstrates how to work with two linked tables. One table holds the IDs and names of groups, another holds records kept for groups. The structure of the tables and the data are stored in XML files (scheme.xml and datasets.xml accordingly). Initially, there is one group and a single record for it. The application is an .aspx page that displays both tables and lets you edit their contents: add, rename and remove groups, add records to groups, edit and delete them (singly or all at once).

You can find the sample projects in the following folders:

- Microsoft Visual Studio .NET 7.x projects:
  - <AQtime>\Samples\Managed\VS.NET\ASP.NET\MasterDetails\VB\ - Microsoft Visual Basic .NET
  - <AQtime>\Samples\Managed\VS.NET\ASP.NET\MasterDetails\CS\ - Microsoft Visual C# .NET
  - <AQtime>\Samples\Managed\VS.NET\ASP.NET\MasterDetails\JS\ - Microsoft Visual J# .NET
- Microsoft Visual Studio 2005 projects:
  - <AQtime>\Samples\Managed\VS2005\ASP.NET\MasterDetails\VB\ - Microsoft Visual Basic .NET
  - <AQtime>\Samples\Managed\VS2005\ASP.NET\MasterDetails\CS\ - Microsoft Visual C# .NET
  - <AQtime>\Samples\Managed\VS2005\ASP.NET\MasterDetails\JS\ - Microsoft Visual J# .NET

You can download compiled versions of these sample applications from http://www.automatedqa.com/downloads/aqtime/.

Profiling ASP.NET Applications via ASP.NET Development Server

This topic describes how to profile ASP.NET applications or services using the ASP.NET Development Server. The server is integrated with Microsoft Visual Studio 2005 and Visual Studio 2008 and can be easily applied while creating, debugging and profiling ASP.NET applications.

Currently, it is impossible to profile a 64-bit ASP.NET application via Visual Studio's ASP.NET Development Server because the server (WebDev.WebServer.exe) is an x86 process. You can run and profile 64-bit ASP.NET applications with IIS. See *Profiling ASP.NET Applications via Internet Information Services* for instructions on how to do this.

The topic contains the following sections:

- *Sample Application*
- *Requirements*
- *Compiling the Application*
- *Profiling the Application*
- *Using the Allocation Profiler*

Sample Application

In our description we will use a sample application called *XMLTrace*. It parses an XML file and displays its contents and structure. You can find the application sources in the following folders:

- *<AQtime>\Samples\Managed\VS2005\ASP.NET\XMLTrace\VB* - Microsoft Visual Basic 2005 and 2008 version
- *<AQtime>\Samples\Managed\VS2005\ASP.NET\XMLTrace\CS* - Microsoft Visual C# 2005 and 2008 version
- *<AQtime>\Samples\Managed\VS2005\ASP.NET\XMLTrace\JS* - Microsoft Visual J# 2005 and 2008 version

You can download the compiled versions of these sample applications from http://www.automatedqa.com/downloads/aqtime/.

We will use the Visual Basic 2005 sample located in the *<AQtime>\Samples\Managed\VS2005\XMTrace\VB* folder. To run this sample application, you will need Microsoft Internet Explorer 5.5 or later.

Requirements

Profiling ASP.NET applications requires 32- or 64-bit editions of Windows 7, Windows Server 2008, Windows Vista, Windows XP, Windows Server 2003 or Windows 2000 with Internet Information Services ver. 4.0 - 7.0.

Compiling the Application

Open the sample project *<AQtime>\Samples\Managed\VS2005\ASP.NET\XMLTrace\VB\XMLTrace.sln* in Visual Studio and build or publish it. See *How AQtime Profilers Use Metadata and Debug Information* to decide whether to include debug information.

To build the project, select **Build | Build Web Site** from the Visual Studio menu.

To publish it, select the **Build | Publish Web Site** menu item and uncheck the "Allow this precompiled site to be updatable" box in the ensuing Publish Web Site dialog.

When you compile an ASP.NET application, Visual Studio organizes the application code into several temporary dynamic link libraries whose location and names are changed from one compilation to another. The library name includes the source file name and some hash value and looks like `App_Web_xmltrace.aspx.cdcab7d2.dll`. Since the names are changed, you have to include new modules in your AQtime project every time you compile your application, and there is no guarantee that you will add all application modules, since their names and locations are defined by Visual Studio.

To avoid these problems, you can publish your ASP.NET project before profiling it (you should publish your ASP.NET application every time you compile it). The folder that you will specify for publishing will contain all DLLs generated for your ASP.NET application. It is recommended that you publish your project to the folder that is used as a virtual directory. If you publish it to another folder, you should specify that folder as a source folder for your virtual directory.

If you do not publish the project, you can still profile all application modules. To do this, add any module to your AQtime project and check the *Profile Entire .NET Code* option in the Setup panel.

Profiling the Application

To profile your ASP.NET application with AQtime using the ASP.NET Development Server:

- Open the sample project *<AQtime>\Samples\Managed\VS2005\ASP.NET\XMLTrace\VB\XMLTrace.sln* in Visual Studio and build or publish it. See *How AQtime Profilers Use Metadata and Debug Information* to decide whether to include debug information.

  To build the project, select **Build | Build Web Site** from the Visual Studio menu.

  To publish it, select the **Build | Publish Web Site** menu item and uncheck the "Allow this precompiled site to be updatable" box in the ensuing Publish Web Site dialog.
- Open the application's library modules from the folder that you published your application to in AQtime's Setup panel. If you did not publish your ASP.NET project, enable the *Profile Entire .NET Code* option.
- Select **Normal** from the **Profiling Mode** dropdown list box on AQtime's **Standard** toolbar (or on Visual Studio's **AQtime** toolbar).
- In AQtime running as a standalone application, select **Run | Parameters** from the main menu. In AQtime integrated into Visual Studio, select **Profile | Parameters** from the Visual Studio menu. Specify an ASP.NET Development Server as a host application and define command line parameters to it in the Run Parameters Dialog (for Normal Mode). The server host application is: *<Windows>\Microsoft.NET\Framework\<Framework_Version>\WebDev.WebServer.exe*. A command line defines the port number, physical path and virtual path of the profiled ASP.NET application and has the following syntax: */port:<port_number> /path:<physical_path> [/vpath:<virtual_path>]* where:
  - <*port_number*> - the number of an unused port, between 1 and 65535.
  - <*physical_path*> - a valid directory name where the ASP application is located.
  - <*virtual_path*> - [optional] a virtual path or application root in the form '/<app name>'. The default value is '/'.

  For instance, for the sample project the command line would be: */port:1169 /path:"C:\Work\AQtime\Samples\Managed\VS2005\ASP.NET\XMLTrace" /vpath:"/XMLTrace"*
- Select the desired profiler and press **Run** to start profiling. In AQtime integrated into Visual Studio, select the desired profiler in the **Profile | Profiler** menu item or from the **Profilers** box on the **AQtime** toolbar; then choose **Profile | Run** (or select **Debug | Run** while one of AQtime panels is active) to start profiling.

AQtime may ask your permission to restart the IIS Admin service. Answer "Yes" to this question.

- Open the *http://localhost/XMLTrace/VB/xmltrace.aspx* file in Internet Explorer and perform profiling as your needs dictate.

  When you are profiling an ASP.NET application, the Event View panel displays events that occur both in the ASP.NET process and in the profiled application. One important thing about this is that all displayed general events occurred in the ASP.NET process, all .NET specific events occurred in the profiled ASP.NET application. (For detailed information on general events and .NET specific events, see *Event View Panel*).

- To obtain the results, select **Run | Get Results** from AQtime's menu. (**Profile | Get Results** from the main menu of Visual Studio). Another way to obtain the results is to create an action that will "tell" AQtime to generate the results. See *Getting Results During Testing* for more information.

- To terminate the profiler run, select **Run | Terminate** from AQtime's main menu or press **Terminate** on the **Standard** toolbar.(or select **Profile | Terminate** from the main menu of Visual Studio).

Note for Windows x64 users: Under 64bit versions of Windows, the ASP.NET process can operate in 64-bit mode (default) or in 32-bit mode. The "bitness" (32-bit or 64-bit) of a process must match the bitness of the ASP.NET application, otherwise it will not run.

Currently, it is impossible to profile a 64-bit ASP.NET application via Visual Studio's ASP.NET Development Server because the server (WebDev.WebServer.exe) is an x86 process and AQtime fails to start profiling when the application's platform target option is set to *Any CPU* (default) or *x64*. To profile an application you should either set the option to *x86* and use ASP.NET Development Server or to profile via the IIS. Note that profiling an application in 32-bit mode under a 64-bit platform produces inaccurate results. It is recommended to use IIS to profile 64-bit ASP applications. See *Profiling ASP.NET Applications via Internet Information Services* for instructions on how to do this.

Using the Allocation Profiler

Your application may contain classes inherited from the `System.Web.UI.Page` or `System.Web.HttpApplication` class. If you include these inherited classes in profiling tasks and profile your .NET application with the **Allocation** profiler, you may note that AQtime reports that no instances of the inherited classes are created.

This happens because the ASP.NET process does not create instances of your classes, which are inherited from `System.Web.UI.Page` or `System.Web.HttpApplication`. It creates instances of temporary classes that are inherited from your classes. Since your classes are not created, the profiling results are empty for them. However, the Allocation profiler automatically includes temporary classes in profiling tasks and traces the creation and deletion of their instances. By viewing profiling results for temporary classes, you can see how the application used your classes during the profiler run.

The names of temporary classes inherited from `System.Web.UI.Page` have the following format:

`ASPXFileName``_aspx`,

where *ASPXFileName* is the name of the .aspx file (excluding the file extension) that refers to your class. For instance, the XMLTrace sample application includes the XMLTrace.aspx file that refers to the `XMLTraceForm` class defined in XMLTrace.aspx.cs. The name of a temporary class inherited from the `XMLTraceForm` class will be `XMLTrace_aspx`.

The names of temporary classes inherited from `System.Web.HttpApplication` have the *`ASAXFileName`*`_asax` format. For example, the XMLTrace application includes the Global.asax file that

holds a reference to the `Global` class defined in Global.asax.cs. The name of a temporary class inherited from `Global` will be `Global_asax`.

Profiling ASP.NET Applications via Internet Information Services

This topic describes how to profile ASP.NET applications or services using the Internet Information Services (IIS). Any ASP.NET application that is created with any development tool can be profiled with the help of the IIS. Applications created in Microsoft Visual Studio 2005 can also be profiled with the ASP.NET Development Server, but there are some difficulties with profiling 64-bit applications. See *Profiling ASP.NET Applications via ASP.NET Development Server* for details.

- *Sample Application*
- *Requirements*
- *Configuring IIS*
- *Preparing Application for Profiling*
- *Running a Profiler*
- *Using the Allocation Profiler*
- *Preparing AQtime Projects When Profiling 64-bit ASP.NET Applications*

Sample Application

In our explanation we will use a sample application called *XMLTrace*. It parses an XML file and displays its contents and structure. You can find the application's sources in the following folders:

- Microsoft Visual Studio .NET 7.x projects:
  - <AQtime>\Samples\Managed\VS.NET\ASP.NET\XMLTrace\VB\ - Microsoft Visual Basic .NET
  - <AQtime>\Samples\Managed\VS.NET\ASP.NET\XMLTrace\CS\ - Microsoft Visual C# .NET
  - <AQtime>\Samples\Managed\VS.NET\ASP.NET\XMLTrace\JS\ - Microsoft Visual J# .NET

You can download compiled versions of these sample applications from http://www.automatedqa.com/downloads/aqtime/.

We will use the Visual Basic .NET sample located in the *<AQtime>\Samples\Managed\VS.NET\XMTrace\VB* folder. To run this sample application, you will need Microsoft Internet Explorer 5.5 or later.

Requirements

Profiling ASP.NET applications requires 32- or 64-bit editions of Windows 7, Windows Server 2008, Windows Vista, Windows XP, Windows Server 2003 or Windows 2000 with Internet Information Services ver. 4.0 - 7.0.

To profile 32-bit ASP.NET modules on a 32-bit operating system, you use AQtime x86.

To profile 64-bit ASP.NET modules on a 64-bit operating system, you use AQtime x64.

To profile 32-bit ASP.NET modules on a 64-bit operating system, you can use either AQtime x86, or AQtime x64. However, you should enable the support for 32-bit applications in the IIS and ASP.NET settings. For more information on this, see below.

Configuring IIS

In further explanations, we will assume that ASP.NET and IIS are running in the default configuration. If ASP.NET or IIS is running under a user account, you should specify certain permissions for this account. Follow the link for more information about this. For more information on this, see *Profiling ASP.NET and IIS Applications: ASP.NET or IIS is Running Under a User Account*.

For 64-bit versions of IIS: Under 64-bit versions of Windows, the ASP.NET process can operate in 64-bit mode (default) or in 32-bit mode. On 64-bit operating systems, 64-bit processes cannot load 32-bit DLLs and 32-bit processes cannot load 64-bit libraries. So, if your IIS application is a 32-bit application and you are going to run it on a 64-bit version of IIS, then you must configure IIS in order for it to be able to run your application.

To switch ASP.NET and IIS to the 32-bit mode:

- Open the command-line window, type the following command and press ENTER:

```
cscript %SystemDrive%\inetpub\AdminScripts\adsutil.vbs SET
W3SVC/AppPools/Enable32BitAppOnWin64 1
```

- Go to the <Windows>\Microsoft.NET\Framework\<Framework_Version> folder. Here, `NET_VERSION` is a placeholder for the folder, whose name corresponds to the .NET Framework version installed on your computer. For instance, it can be `v1.1.4322`, `v2.0.50727` and so on.
- Run the `aspnet_resiis.exe` executable that resides in this folder with the -i argument in the command line:

```
aspnet_resiis.exe -i
```

To switch ASP.NET and IIS back to the 64-bit mode:

- Open the command-line window, type the following command and press ENTER:

```
cscript %SystemDrive%\inetpub\AdminScripts\adsutil.vbs SET
W3SVC/AppPools/Enable32BitAppOnWin64 0
```

- Go to the *<Windows>\Microsoft.NET\Framework64\<Framework_Version>* folder (note the 64 postfix in the folder name) and run the `aspnet_resiis.exe` executable that resides in this folder:

```
aspnet_resiis.exe -i
```

Preparing Application for Profiling

- Open the sample project *<AQtime>\Samples\Managed\VS.NET\ASP.NET\XMLTrace\VB\XMLTrace.sln* in Visual Studio and compile it there. See see *How AQtime Profilers Use Metadata and Debug Information* to decide whether to include debug information.

  For Visual Studio 2005 or 2008 users: When you compile an ASP.NET application, Visual Studio organizes the application code into several temporary dynamic link libraries whose location and names are changed from one compilation to another. Since the names are changed, you have to include new modules in your AQtime project every time you compile your application, and there is no guarantee that you will add all application modules, since their names and locations are defined by Visual Studio.

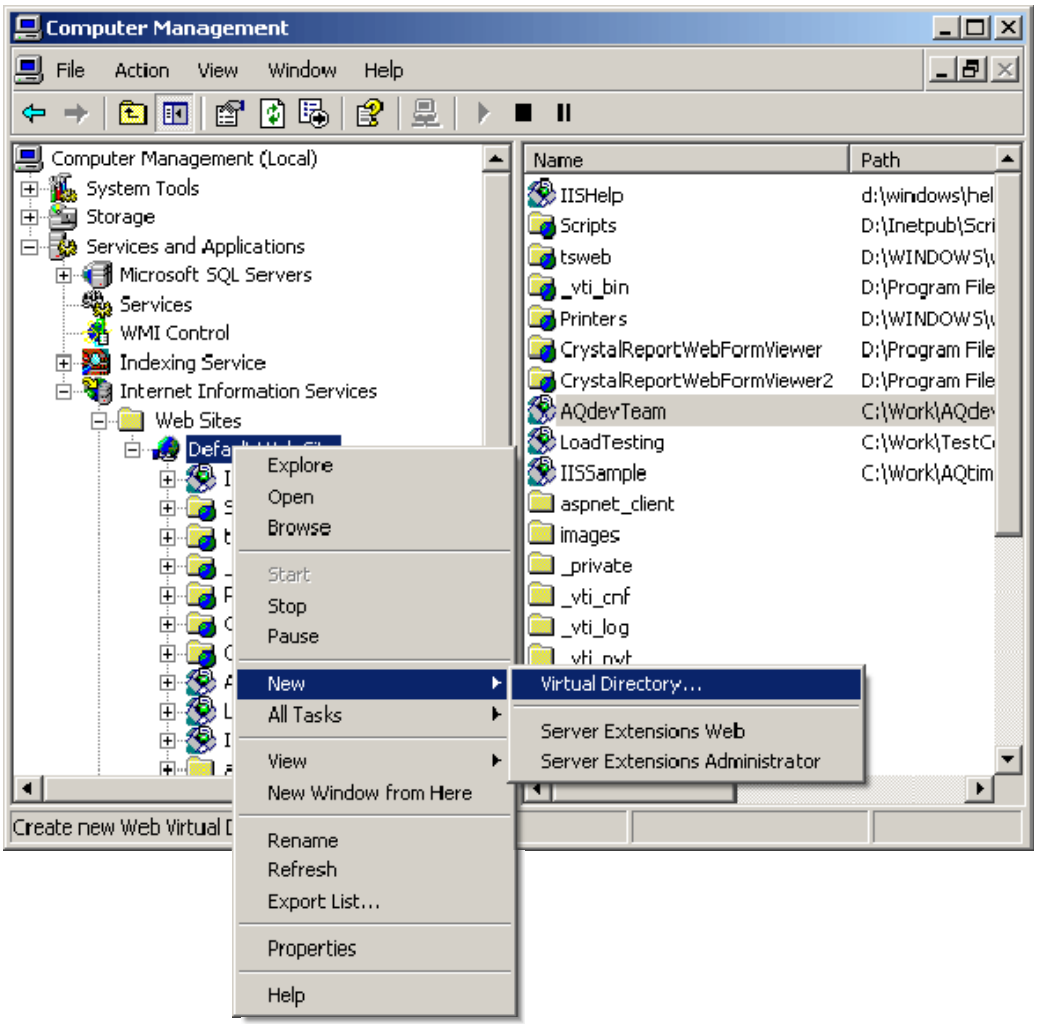
To avoid these problems, you can publish your ASP.NET project before profiling it (you should publish your ASP.NET application every time you compile it). The folder that you will specify for publishing will contain all DLLs generated for your ASP.NET application. It is recommended that you publish your project in the folder that is used as a virtual directory. If you publish it to another folder, you should specify that folder as a source folder for your virtual directory.

For Visual Studio .NET 7.x users: The XMLTrace application compiled with Visual Studio .NET 7.x requires that the System.Web.dll library be in the application's folder. Copy this library from your .NET Framework folder (for example, *<Windows>\Microsoft.NET\Framework\<Framework_Version>*) to the *<AQtime>\Samples\Managed\ASP.NET\XMLTrace\VB\bin* folder.

- Create an IIS application corresponding to our sample ASP.NET application. The way you do this depends on the IIS version.

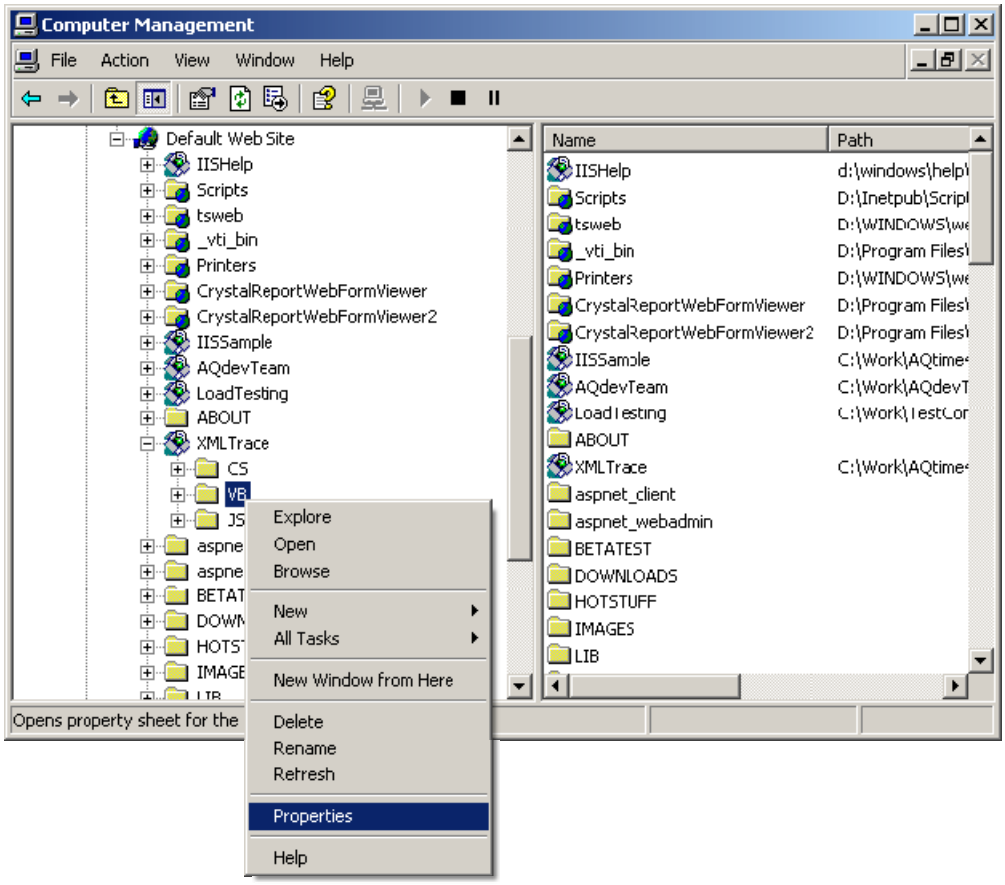
  If you use IIS 4 - 6, follow these steps:

  - Open the **Control Panel | Administrative Tools | Internet Information Services** window (it is displayed in **Microsoft Management Console**).
  - Right-click the *Default Web Site* node and select **New | Virtual Directory** from the context menu.

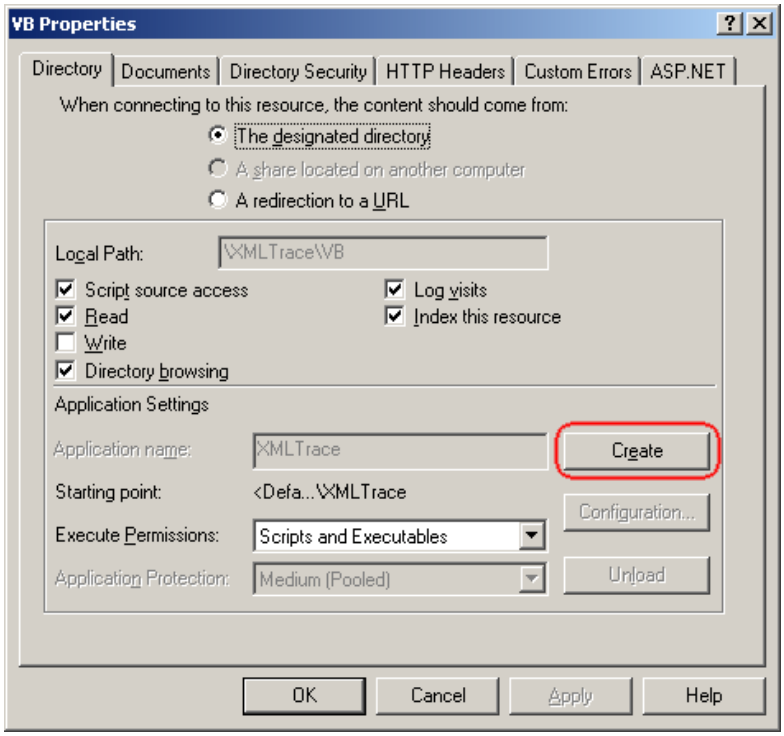


Follow the instructions of the ensuing wizard to create a new virtual directory. Specify *XMLTrace* as the virtual directory alias and map this virtual directory to the folder, where the compiled DLLs reside.

- In the Internet Information Services Manager, right-click the *XMLTrace\VB* node and select **Properties** from the context menu.



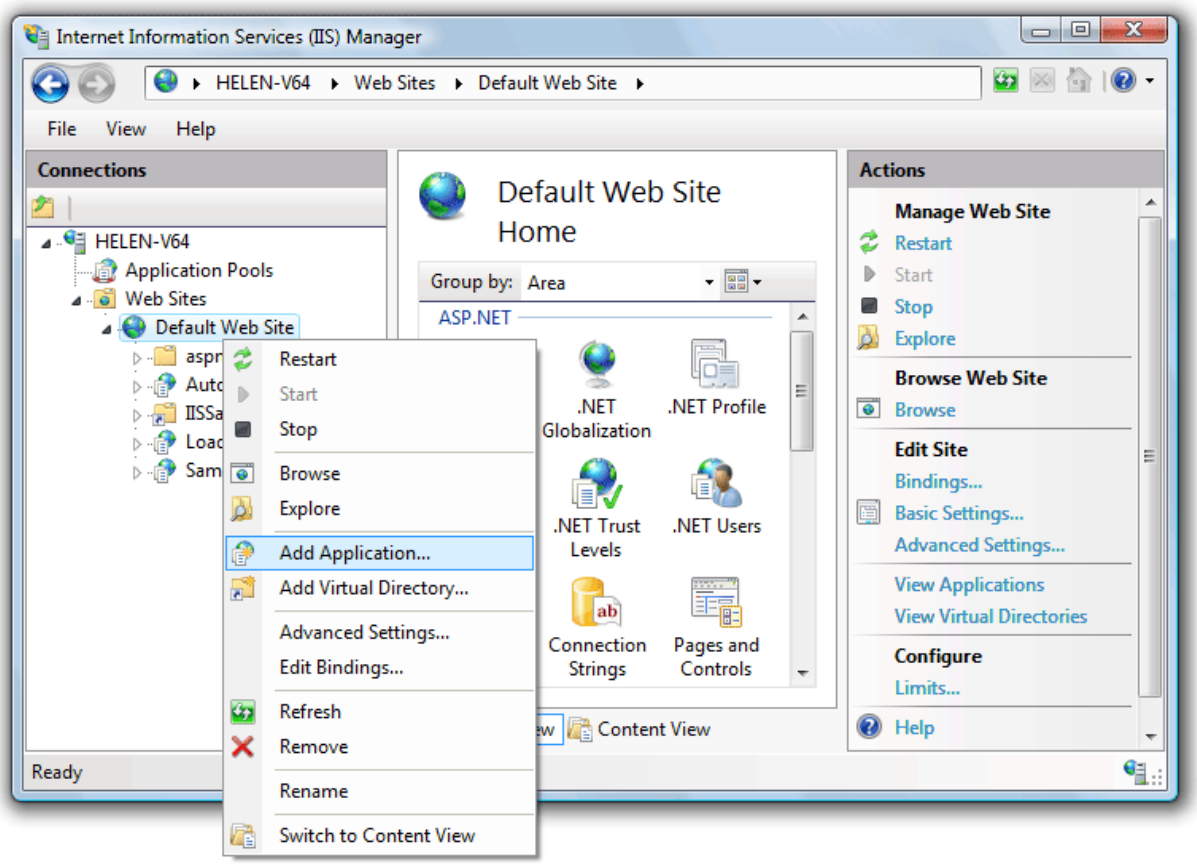
This will call the dialog where you can create and modify properties of our IIS application.



- In the dialog, press **Create**. This will create a new IIS application. Then press **OK** to close the dialog.

If you use IIS 7, follow these steps:

- Open the **Control Panel | Administrative Tools | Internet Information Services (IIS) Manager** window.
- Right-click the *Default Web Site* node and select **Add Application** from the context menu.



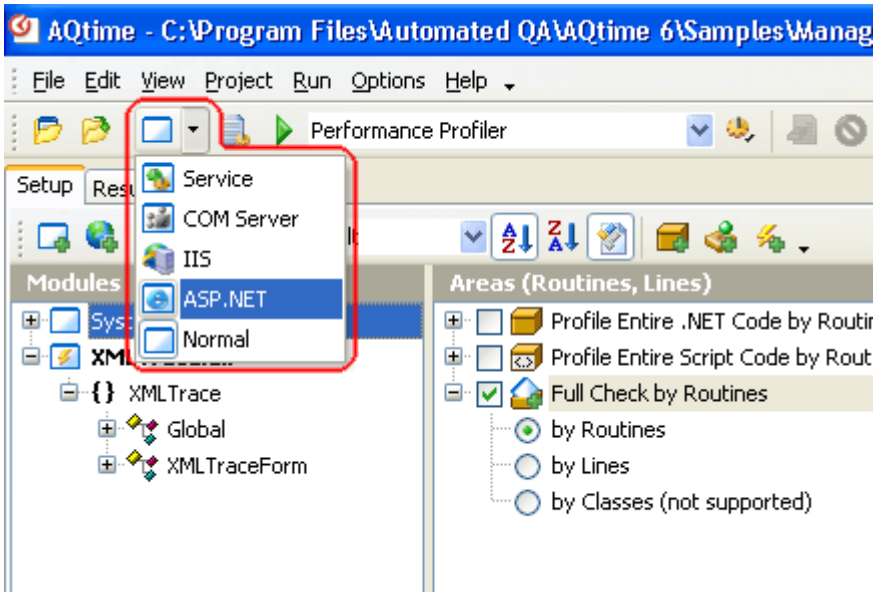
Use the ensuing **Add Application** dialog to create a new IIS application. Specify *XMLTrace* as the application alias and map it to the folder where the compiled DLLs reside.

Running a Profiler

Now we can profile our ASP.NET application with AQtime:

- Load your application's library in AQtime. The `xmltrace.dll` file is in the *XMLTrace\VB\bin* folder, if you compiled the application in Visual Studio .NET. If you compiled the sample in Visual Studio 2005 or 2008, the module name will differ: it will include the source file name and a hash value, for instance, it can be `App_Web_xmltrace.aspx.cdcad7d2.dll`.

- Select **ASP.NET** from the Profiling Mode dropdown list box on AQtime's **Standard** toolbar (on the **AQtime** toolbar in Visual Studio or on the **Run.AQtime** toolbar in Borland Developer Studio).



- In AQtime running as a standalone application, select the desired profiler and click **Run** to start profiling. In AQtime integrated into Visual Studio, select the desired profiler in the **Profile | Profiler** menu item or from the **Profilers** box on the **AQtime** toolbar; then choose **Profile | Run** to start profiling. In AQtime integrated into Borland Developer Studio, select the desired profiler in the **Profile | Current Profiler** menu item; then choose **Run | Run With Profiling** to start profiling. AQtime may ask your permission to restart the IIS Admin service. Answer "Yes" to this question.
- Open the *http://localhost/XMLTrace/VB/xmltrace.aspx* file in the Internet Explorer and perform profiling as your needs dictate.

  To avoid launching Internet Explorer manually, you can specify the desired Web page in the **Start Page** box in the Run Parameters Dialog (for ASP.NET Mode). In this case, AQtime will automatically open this page in Internet Explorer once you start profiling.

  When you profile an ASP.NET application, the Event View panel displays both events that occur in the ASP.NET process and in the profiled application. One important thing about this is that all displayed general events occurred in the ASP.NET process, all .NET specific events occurred in the profiled ASP.NET application. (For detailed information on general events and .NET specific events, see *Event View Panel*).

  Note for Windows x64 users: Under 64bit versions of Windows, the ASP.NET process can operate in 64-bit mode (default) or 32-bit mode. The "bitness" (32-bit or 64-bit) of a process must match the bitness of the ASP.NET application, otherwise it will not run.

  When working with the Internet Information Services, AQtime checks the profiled ASP.NET application and automatically selects the proper mode of the ASP.NET process.

  For information on how to switch between 32- and 64-bit versions of ASP.NET, see the

following article from Microsoft Knwoledge Base: *How to switch between the 32-bit versions of ASP.NET 1.1 and the 64-bit version of ASP.NET 2.0 on a 64-bit version of Windows* (http://support.microsoft.com/?id=894435).

In the default configuration, IIS works as a service. It does not close when you close Internet Explorer, so AQtime does not generate any profiling results. To obtain the results, select **Run | Get Results** from AQtime's menu (select **Profile | Get Results** in Visual Studio or click **Get Results** on the **Run.AQtime** toolbar in Borland Developer Studio). Another way to obtain the results is to create an action that will tell AQtime to generate the results. See *Getting Results During Testing* for more information.

- To terminate the profiler run, select **Run | Terminate** from AQtime's main menu or press **Terminate** on the Standard toolbar (select **Profile | Terminate** from the main menu of Visual Studio or click **Terminate** on the **Run.AQtime** toolbar in Borland Developer Studio).

Using the Allocation Profiler

Your application may contain classes inherited from the `System.Web.UI.Page` or `System.Web.HttpApplication` classes. If you include these inherited classes in profiling tasks and profile your .NET application with the **Allocation** profiler, you may note that AQtime will report that no instances of the inherited classes were created.

This happens because the ASP.NET process does not create instances of your classes, which are inherited from `System.Web.UI.Page` or `System.Web.HttpApplication`. It creates instances of temporary classes that are inherited from your classes. Since your classes are not created, the profiling results for them are empty. However, the Allocation profiler automatically includes temporary classes in profiling tasks and traces the creation and deletion of their instances. By viewing profiling results for temporary classes, you can see how the application used your classes during the profiler run.

The names of temporary classes inherited from `System.Web.UI.Page` have the following format:

`ASPXFileName``_aspx`,

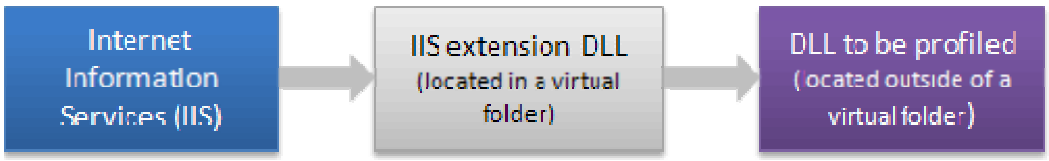
where *ASPXFileName* is the name of the .aspx file (excluding the file extension) that references your class. For instance, the XMLTrace sample application includes the XMLTrace.aspx file that refers to the `XMLTraceForm` class defined in XMLTrace.aspx.cs. The name of a temporary class inherited from the `XMLTraceForm` class will be `XMLTrace_aspx`.

The names of temporary classes inherited from `System.Web.HttpApplication` have the *`ASAXFileName`*`_asax` format. For example, the XMLTrace application includes the Global.asax file that holds a reference to the `Global` class defined in Global.asax.cs. The name of a temporary class inherited from `Global` will be `Global_asax`.

Preparing AQtime Projects When Profiling 64-bit ASP.NET Applications

AQtime can profile both 32- and 64-bit ASP.NET applications. These applications are opened either with a 32- or a 64-bit IIS process. When you start profiling in ASP.NET mode, AQtime attempts to detect the "bitness" of this process automatically. To do this, AQtime finds virtual folders that contain modules added to the Setup panel and then detects the "bitness" of IIS application pools in which these modules are used.

However, this approach does not work if you are profiling a module that does not belong to any virtual folder:


Internet Information Services (IIS)
IIS extension DLL (located in a virtual folder)
DLL to be profiled (located outside of a virtual folder)

If your project contains only those modules that do not belong to a virtual folder, AQtime is unable to detect the "bitness" automatically and always works with a 32-bit IIS process. To work around this problem, include a DLL, which is located in a virtual folder, in your AQtime project. This will help AQtime determine the "bitness" of the IIS process. There is no need to include it in profiling tasks if you do not need this. The only purpose of this DLL is to help AQtime.

Profiling IIS Applications

This topic describes profiling of Internet Information Services (IIS) applications created with unmanaged (ordinary) compilers. Using .NET compilers provided by Microsoft Visual Studio .NET 2002, Visual Studio .NET 2003 or Visual Studio 2005, you can create IIS applications that use .NET assemblies. For more information on how to profile these applications, see *Profiling ASP.NET Applications*. To learn how to profile NT services, review *Profiling Services*.

Requirements

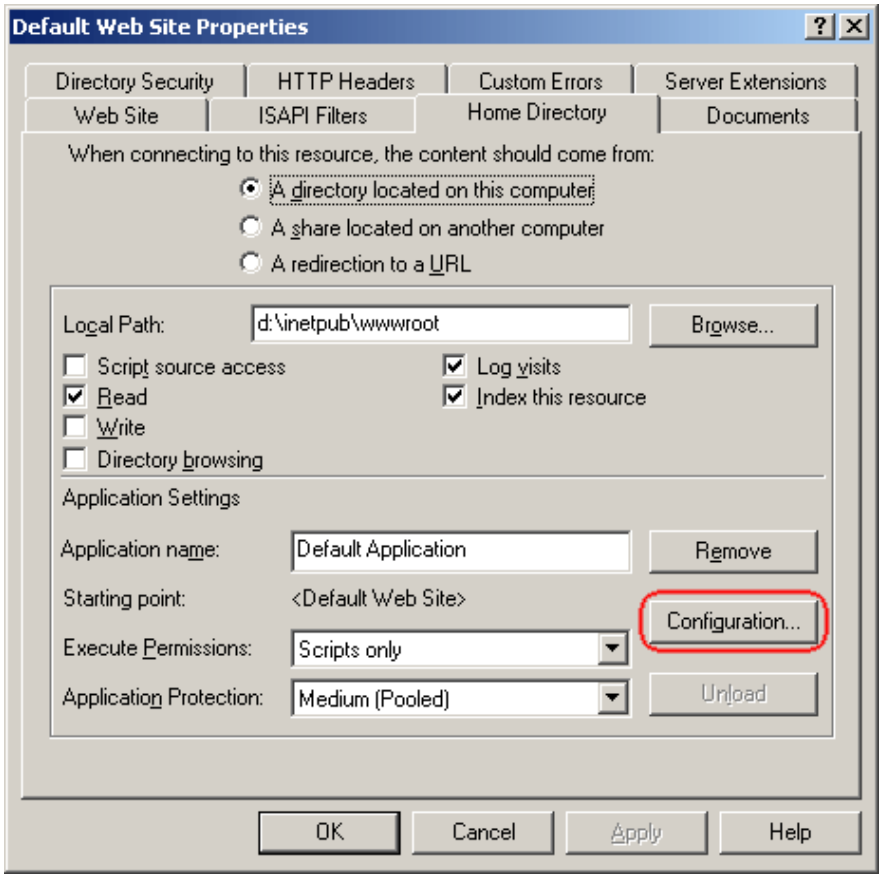
- Profiling IIS applications requires Windows 2000, Windows XP (either 32-, or 64-bit edition), Windows Server 2003 (either 32-, or 64-bit edition) Windows Vista (either 32-, or 64-bit edition), Windows Server 2008 (either 32-, or 64-bit edition) or Windows 7 (either 32-, or 64-bit edition) and Internet Information Services ver. 4.0 - 7.0.
- Currently, AQtime can profile only those IIS applications, whose "bitness" corresponds to the "bitness" of the operating system. That is, if you run AQtime x64 on a 64-bit operating system, it will only be able to profile 64-bit applications. If you run AQtime x86 on a 32-bit operating system, it will only be able to profile 32-bit applications. Currently, when AQtime x86 runs on a 64-operating system it cannot profile 32-bit IIS applications.

Configuring IIS

In our explanations, we assume that IIS is running in the default configuration. If it is running under a user account, you should specify certain permissions for this account. For more information on this, see *Profiling ASP.NET and IIS Applications: ASP.NET or IIS is Running Under a User Account*.

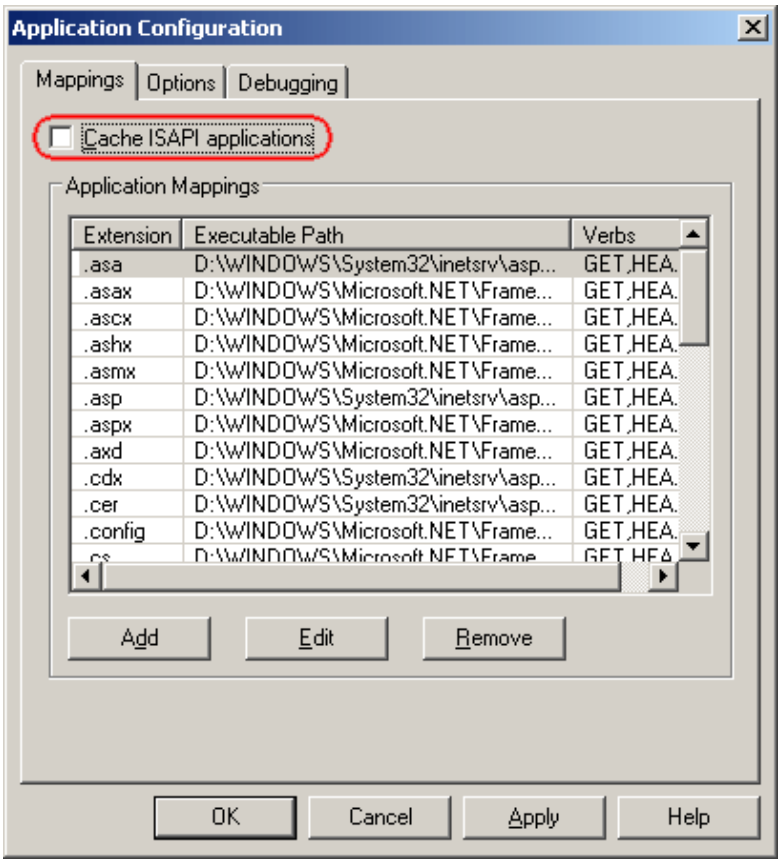
If you use Internet Information Services ver. 4 - 6, disable the *Cache ISAPI applications* setting of IIS:

- Open the Internet Information Services Manager dialog (Control Panel | Administrative Tools | Internet Information Services).
- Right-click the root node of your Web site and select **Properties** from the context menu. This will open the **Properties** dialog for the root virtual directory.
- In the dialog, switch to the **Home Directory** page and press **Configuration** there:



The **Application Configuration** dialog will appear.

- Uncheck the **Cache ISAPI applications** box there and click **OK** to save changes:



For 64-bit versions of IIS: On 64-bit operating systems, 64-bit processes cannot load 32-bit DLLs and 32-bit processes cannot load 64-bit libraries. So, if your IIS application is a 32-bit application and you are going to run it on a 64-bit version of IIS, then you must configure IIS in order for it to be able to run your application.

To do this, open the command-line window, type the following command and press ENTER:

```
cscript %SystemDrive%\inetpub\AdminScripts\adsutil.vbs SET
W3SVC/AppPools/Enable32BitAppOnWin64  1
```

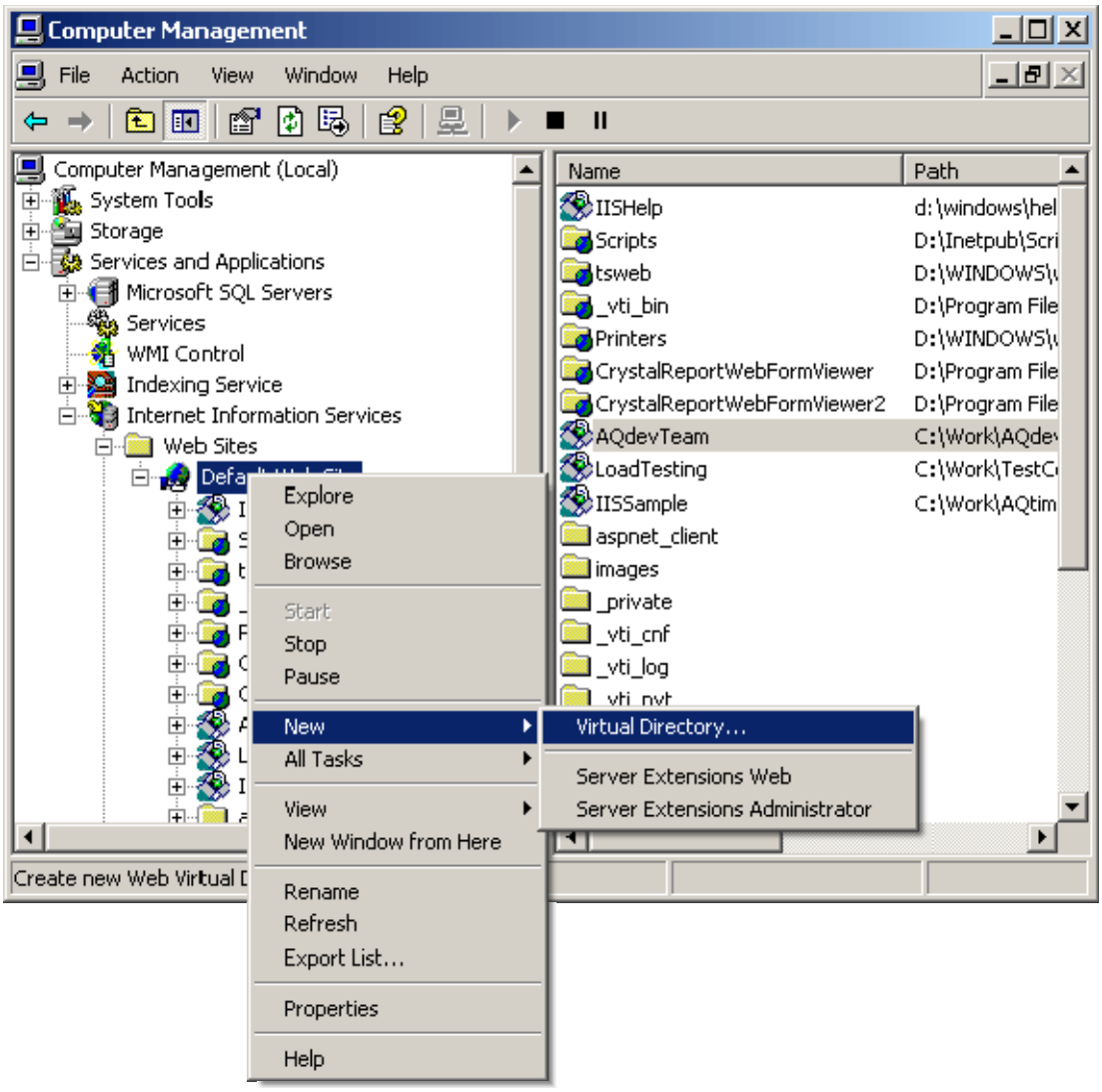
Preparing Application for Profiling

Now let's prepare the application for profiling:

- Open your application in your development tool and compile it with debug information (see *How AQtime Profilers Use Metadata and Debug Information*).

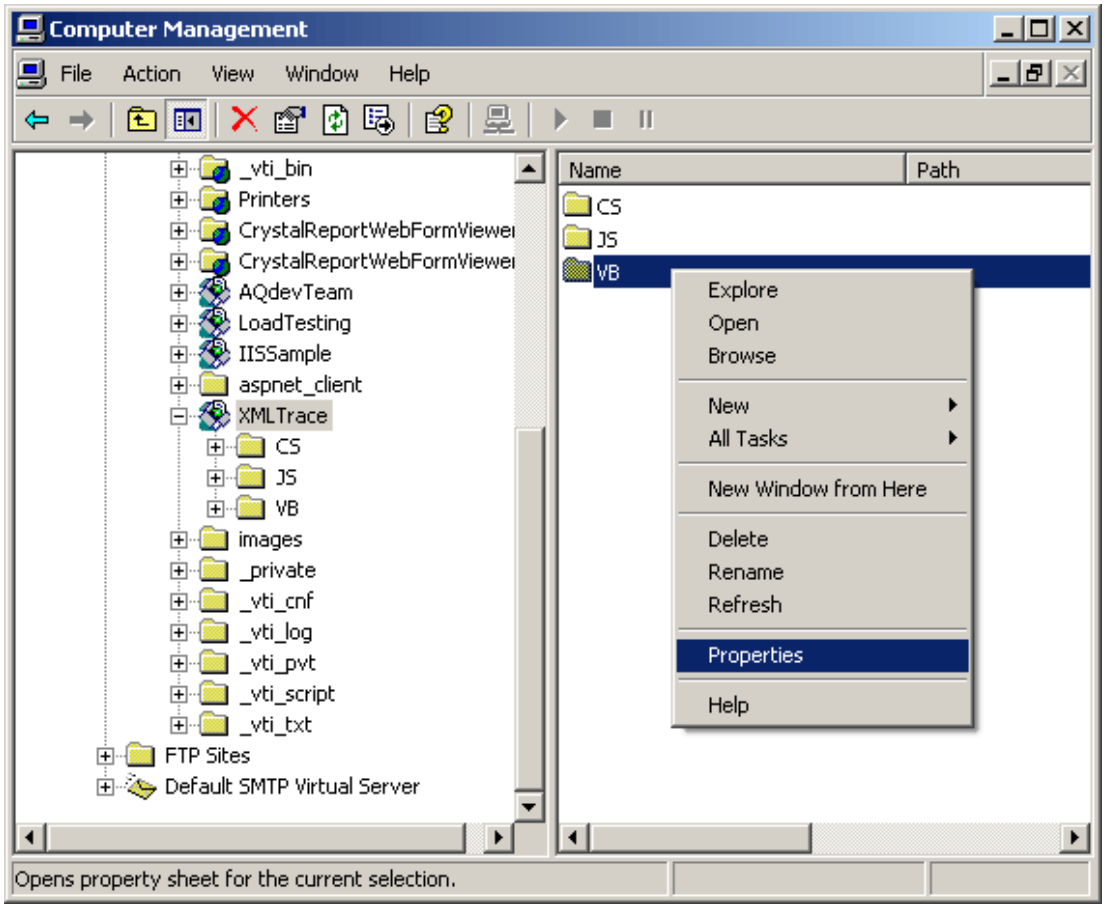
If you use IIS 4 - 6, follow these steps:

- Open the Internet Information Services Manager dialog (Control Panel | Administrative Tools | Internet Information Services).
- Create a virtual directory for your IIS application. To do this, right-click the **Default Web Site** node and select **New | Virtual Directory** from the context menu:

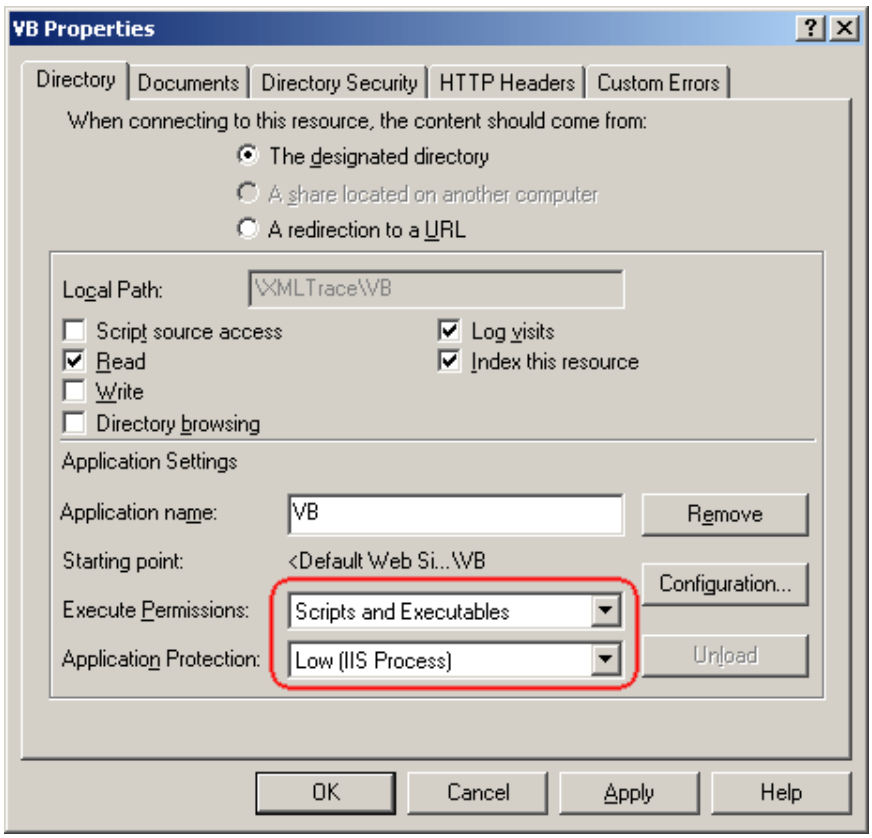


Then follow the instructions of the ensuing dialog.

- In the Internet Information Services Manager, right-click your virtual directory and select **Properties** from the context menu.

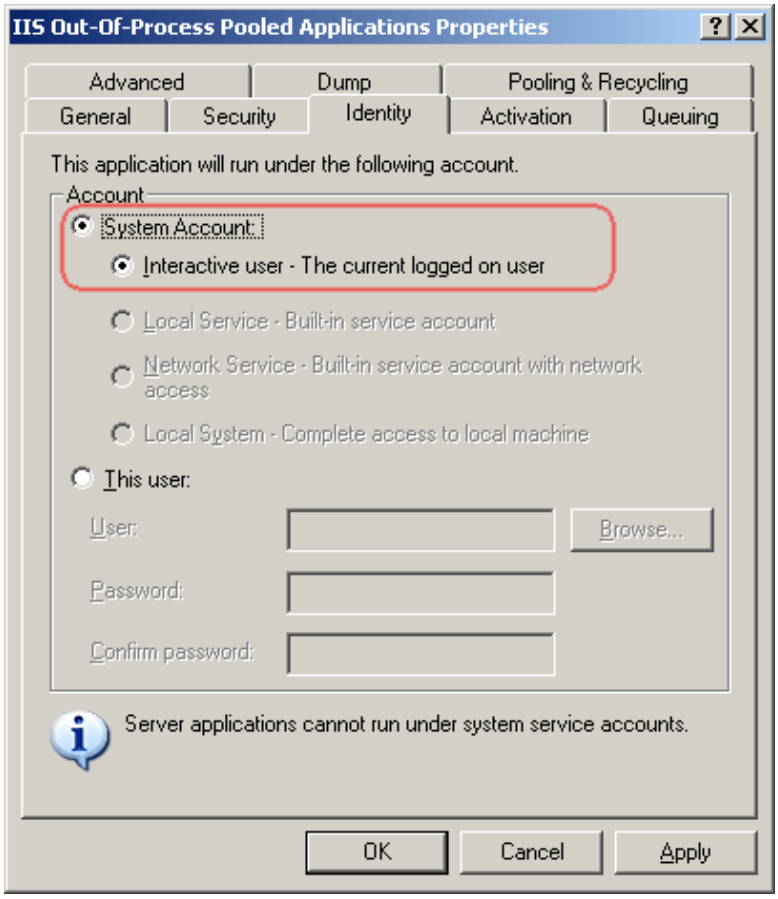


- The **Properties** dialog will appear. In the dialog,
  - Switch to the **Directory** tabbed page.
  - Select *Scripts and Executables* in the **Execute permissions** dropdown list.



- Remember or write down the value of the **Application Protection** option of your IIS application. We will need it later.
- Press **OK** to save changes.

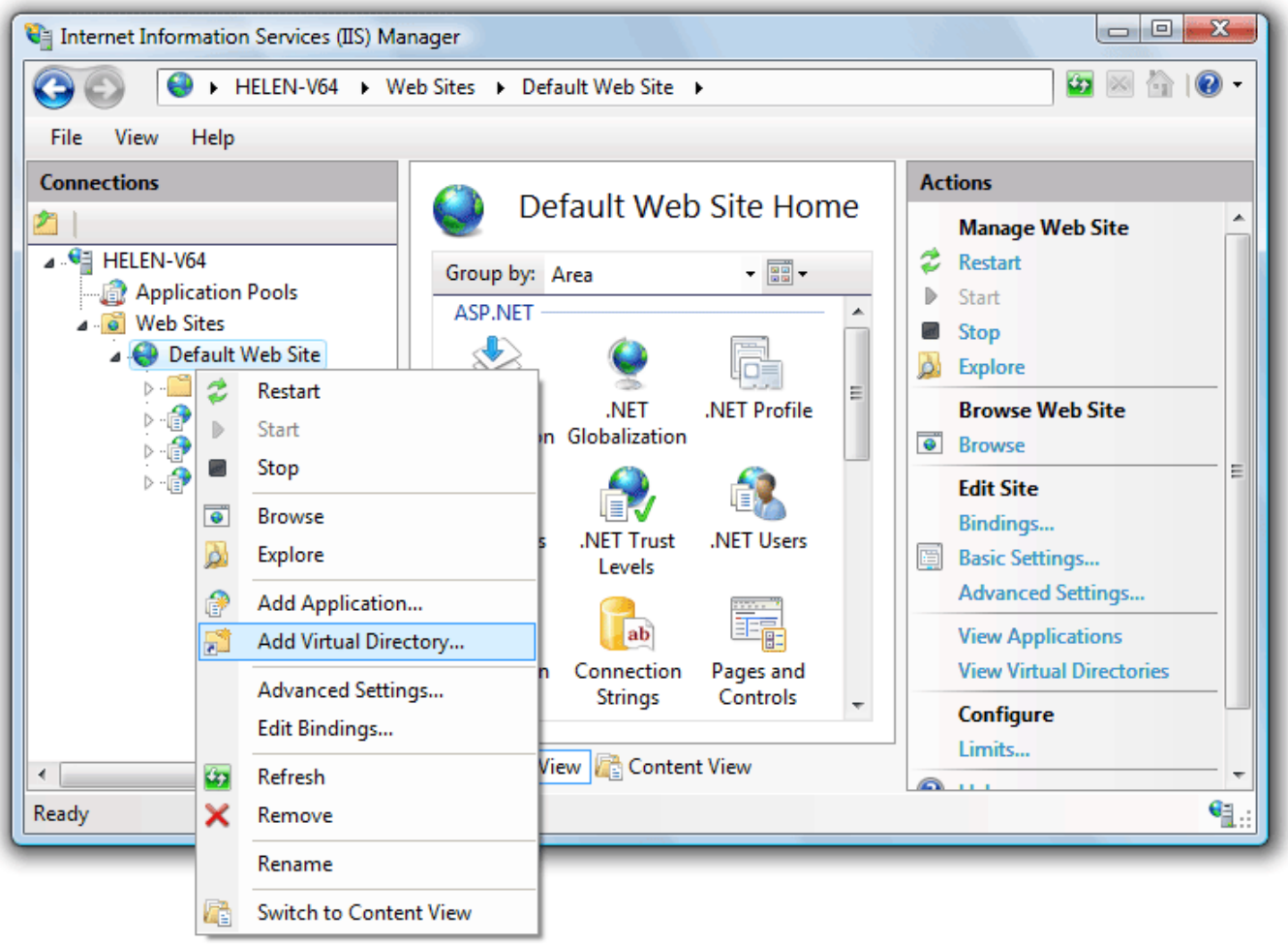
- If you set the **Application Protection** option in the Properties dialog to *High (Isolated)* or *Medium (Pooled)*, you should modify properties that specify the user account, under which the IIS process will run. To do this:
  - Open Control Panel | Administrative Tools | Component Services.
  - Expand the tree down to the node Console Root | Component Services | Computers | My Computer | COM+ Applications | IIS Out-of-Process Pooled Applications.
  - Right-click the IIS Out-of-Process Pooled Applications node and select Properties from the context menu. The IIS Out-of-Process Pooled Applications Properties dialog will appear.
  - Switch to the Identity page of this dialog and set the *Account* option to *Interactive user*.



- Close the dialog by pressing **OK**.

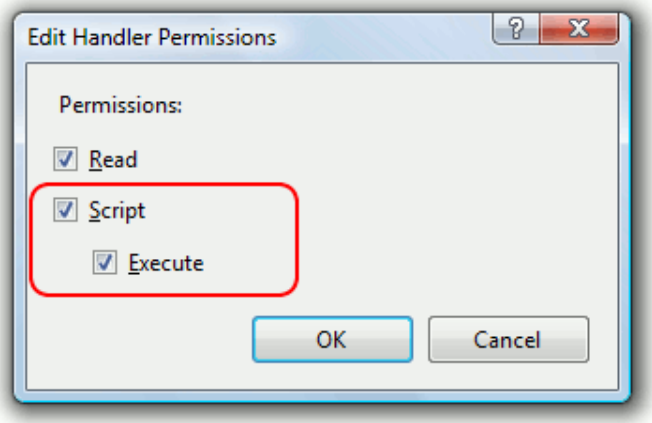
If you use IIS 7, follow these steps:

- Open the Internet Information Services Manager dialog (Control Panel | Administrative Tools | Internet Information Services; Control Panel | Administrative Tools | Internet Information Services (IIS) Manager).
- Create a virtual directory for your IIS application. To do this, right-click the **Default Web Site** node and select **Add Virtual Directory** from the context menu:

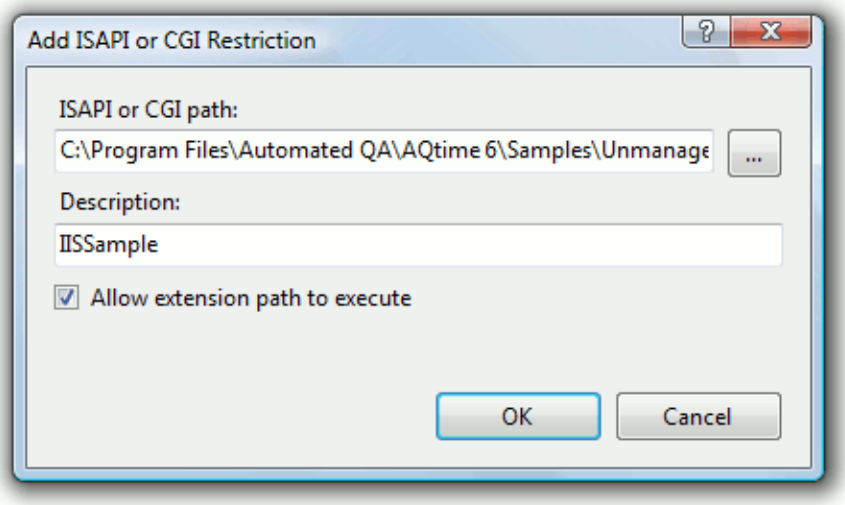


Then follow the instructions of the ensuing dialog.

- In the Internet Information Services Manager, select your virtual directory and double-click the **Handler Mappings** item in the **Features View**.
- Click Edit Handler Permissions in the Actions pane. The Edit Handler Permissions dialog appears.
- In the dialog, check the **Script** and **Execute** options and press **OK** to save the changes.



- If your IIS application includes dynamic content provided by .exe or .dll files, you need to configure ISAPI and CGI Restrictions that will enable their execution on the server. To do this:
  - In the Internet Information Services Manager, select the node corresponding to your web server (this is the top-most node in the **Connections** tree).
  - Double-click **ISAPI and CGI Restrictions** in the **Feature View**.
  - Click **Add** in the **Actions** pane. The **Add ISAPI or CGI Restriction** dialog appears.
  - In the dialog:
    - Specify the full path to the .dll or .exe file in the **ISAPI or CGI Path** box.
    - Enter any descriptive text for the ISAPI or CGI restriction in the **Description** box.
    - Check **Allow the extension path to execute**:



    - Press **OK** to close the dialog and add the specified restriction.

Running a Profiler

Now we can profile our IIS application with AQtime:

- Load your application in AQtime.
- Select **IIS** from the **Profiling Mode** dropdown list box on AQtime's **Standard** toolbar (on the **AQtime** toolbar if you use AQtime integrated into Visual Studio; on the **Run.AQtime** toolbar if you use AQtime integrated into Borland Developer Studio).
- Select **Run | Parameters** from AQtime's main menu (select **Profiler | Parameters** in Visual Studio or click **Parameters** on the **Run.AQtime** toolbar in Borland Developer Studio). This will call the Run Parameters Dialog (for IIS Mode).
- In the dialog's **Start Page** edit box, you can specify the Web page that loads your IIS application. AQtime will automatically open this page in Internet Explorer once you start profiling. If you do not specify the start page in this dialog, you will have to open the desired page in Internet Explorer manually.
- In AQtime running as a standalone application, select the desired profiler and click **Run** to start profiling. In AQtime integrated into Visual Studio, select the desired profiler in the **Profile | Profiler** menu item or from the **Profilers** box on the **AQtime** toolbar; then choose **Profile | Run** to start profiling. In AQtime integrated into Borland Developer Studio, select the desired profiler in

the **Profile | Current Profiler** menu item; then choose **Run | Run With Profiling** to start profiling.

- AQtime will display a message box notifying you that the IISAdmin service should be restarted. Click **Yes** to restart the service.
- If you do not specify the start page in the Run Parameters dialog, AQtime will wait for the start of the process that will use your DLL (IIS applications are normally DLLs). This process can be either the IIS process (inetinfo.exe) or a helper process, which the IIS uses to load your application (for example, dllhost.exe or w3wp.exe). For which process AQtime waits depends on the IIS version you use and on the Application Protection option of your IIS application.
- To start the process, open the page that loads your IIS application, in Internet Explorer. Note that you can avoid manual opening of the page by specifying it in the **Start Page** edit field of the Run Parameters Dialog (for IIS Mode) dialog.
- Now you can perform profiling as your needs dictate.

  In the default configuration, IIS works as a service. It does not close when you close Internet Explorer, so AQtime does not generate any profiling results. To obtain the results, select **Run | Get Results** from AQtime's menu (select **Profile | Get Results** in Visual Studio or click **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar). Another way to obtain the results is to create an action that will tell AQtime to generate the results. See *Getting Results During Testing* for more information.

- To terminate the profiler run, select **Run | Terminate** from AQtime's main menu (select **Profile | Terminate** from the main menu of Visual Studio or click **Terminate** on the **Run.AQtime** toolbar in Borland Developer Studio).

AQtime includes an example of IIS applications - *IISSample*. It demonstrates how you can profile IIS applications with AQtime. For more information on this, see *Profiling IIS Applications - Tutorial* in on-line help. You can find the sample in the following folder:

- *<AQtime>\Samples\Unmanaged\IISSample* - Borland Delphi

You can download a compiled version of this sample application from http://www.automatedqa.com/downloads/aqtime/.

Profiling ASP.NET Applications: ASP.NET or IIS is Running Under a User Account

If either IIS or the ASP.NET process is running under a user account, before profiling your application, you should assign specific permissions to this user to allow them to profile ASP.NET applications:

- Open the **Control Panel | Administrative Tools | Local Security Policy** window.
- Select **Local Policies | User Rights Assignment** from the tree view on the left side of the dialog.
- Ensure that the user name is added to the following lists of security settings:
  - *Act as part of the operating system*
  - *Log on as a service*
  - *Generate security audits*
  - *Replace a process level token*
- After the above settings are selected, restart your computer.

Once you have set the permissions, you can profile your application as it is described in *Profiling ASP.NET Applications* and *Profiling IIS Applications*.

Profiling COM Applications

There are several types of COM servers:

- in-process servers (for example, ActiveX controls).
- out-of-process servers (that is, OLE servers that are executed as a separate process).
- DCOM (Distributed Component Object Model) servers are out-of-process OLE servers that support remote procedure calls, that is, clients that are on other machines in a network.
- COM+ applications (COM+ is the further evolution of COM).
- MTS (Microsoft Transaction Server) is a set of libraries that serve for easier development and deployment of server applications built with the COM technology.

Different types of COM servers are profiled in different ways. For detailed instructions on how to profile them, follow the topics listed below. Before you proceed, we recommend that you read the notes below this list.

- *Profiling In-Process COM Servers*
- *Profiling Out-of-Process COM Servers*
- *Profiling DCOM Servers*
- *Profiling COM+ and MTS Applications*

Notes:

- You can profile COM applications in two modes: *Normal* and *COM Server*. **Normal mode** means that you are profiling your COM server as an ordinary (non-COM) application. That is, the COM server is started under AQtime so that the latter can instrument and profile the server code. However, it is often necessary to profile code that executes when the server is launched by the operating system, not by AQtime. For instance, DCOM servers are launched by the operating system by a request from the client application installed on another computer. Since the servers are started by the OS, AQtime cannot instrument and profile them in Normal mode. To profile COM servers, which client process is not launched by AQtime, use **COM Server mode**. This mode is supported on Windows NT, 2000, XP and Windows Server 2003.

  Profiling in **COM Server mode** requires specifying the client application in the Run Parameters Dialog (for COM Server Mode). The client application name is used to determine the name of the process that will load your COM server. AQtime traces the creation of this process and runs it in debug mode under AQtime (that is, AQtime will act as a debugger). This allows AQtime to instrument the COM server code.

  Note that if you select COM Server mode, AQtime will not launch your COM server application. It waits until this application is launched by a client request. The client application name that you specify in the Run Parameters Dialog (for COM Server Mode) depends on the type of the COM server being tested. For instance, for in-process COM servers it is the name of the COM client application, for DCOM servers, it is dllhost.exe, for out-of-process COM servers, it is the name of the profiled executable. For more information on this, follow the topics listed above.

- A COM server can process several requests from different client applications simultaneously. If you are going to profile a COM server with two or more clients and use the Performance or Coverage profiler, it is recommended that you set the profiler's Thread model option to *COM Threads*. Otherwise, the profilers may trace the parent-child relationship for COM server functions incorrectly. See *Profiling COM Logical Threads* for detailed description of this option.

- Profiling 64-bit COM applications is possible under Windows 64bit platforms.

Profiling In-Process COM Servers

In-process COM servers are usually compiled as DLL or OCX files. They run in the address space of the process that uses the COM server. That is why you should either specify a host application for this DLL (OCX) or add it to the AQtime project that contains an EXE file. You can profile your in-process server either in Normal or in COM server mode. Normal mode can be used if the client application (client process) that will use your in-process COM server is launched by AQtime. If the client process is not launched by AQtime, use COM Server mode. For instance, you can use the COM Server mode when --

- your in-process COM server is started by a DCOM request from another computer (in this case, the client process for your COM server is dllhost.exe launched by the operating system). See also *Profiling DCOM Servers*.
- your server is launched by a request sent from a Web page (in this case, the Internet Information Services process, inetinfo.exe, becomes the client process for your COM server).
- etc.

To profile your in-process COM server in **Normal mode**, follow the steps below:

- Compile your COM application with debug information. See *How AQtime Profilers Use Metadata and Debug Information* for detailed instructions on how to do this.
- Be sure the "debug" version of your COM server is registered in the system. If the server was compiled on your machine, it was registered during compilation. In any case, you can use the regsvr32 utility located in the <Windows>\System32 folder to register the control, for example:

```
regsvr32.exe <Path>\MyServer.DLL
```

- Select **Normal** from the **Profiling Mode** dropdown list box that is displayed on AQtime's **Standard** toolbar (on the **AQtime** toolbar if you use AQtime integrated into Visual Studio; on the **Run.AQtime** toolbar if you use AQtime integrated into Borland Developer Studio).
- Open your COM application in AQtime. You can either
  - Open it as a separate AQtime project. In this case, you will have to specify a host application for the DLL (OCX) in the Run Parameters Dialog (for Normal Mode).

    or
  - Add your DLL (OCX) to the project that holds an EXE file, which will load this DLL (OCX):
    - To add your module to the project, click the **Add Module** item on the toolbar or context menu of the Setup panel.
    - Set the EXE as the "main" module in the project. To do this, right-click EXE in the Setup panel and select **Set as Active Module** from the context menu.

    Note: In 64-bit versions of Windows, 32-bit modules can be loaded into 32-bit processes only and 64-bit modules can be loaded into 64-bit processes only. So, if the "bitness" of your COM server does not match the "bitness" of the process, to which the server is loaded, AQtime will not start profiling.
- Start profiling and perform it in the usual manner. Keep in mind that to profile a function (unit, class) you must check it within a profiling area or select *Profile Entire .NET Code* or *Full Check* to profile all the routines.

To profile your in-process COM server in **COM Server mode**, perform the following steps (note that COM Server mode is supported only on Windows NT, 2000, XP and Windows Server 2003):

- Compile your COM application with debug information. See *How AQtime Profilers Use Metadata and Debug Information* for detailed instructions on how to do this.
- Make sure the "debug" version of your control is registered in the system.
- Open the DLL or OCX module that contains your COM server in AQtime.
- Select **COM Server** from the **Profiling Mode** dropdown list box that is displayed on AQtime's **Standard** toolbar (on the **AQtime** toolbar if you use AQtime integrated into Visual Studio; on the **Run.AQtime** toolbar if you use AQtime integrated into Borland Developer Studio).
- Select **Run | Parameters** from AQtime's main menu (select **Profile | Parameters** in Visual Studio or click **Parameters** on Borland Developer Studio's **Run.AQtime** toolbar). This will open the Run Parameters Dialog (for COM Server Mode). In the dialog:
  - In the **Client Application** box, enter the name of the executable that will load your COM server.
  - Press **OK** to close the dialog.
- Press **Run** to start profiling (**Profile | Run** in Visual Studio or **Run | Run With Profiling** in Borland Developer Studio). AQtime will display a dialog asking you to launch the COM client application.
- Launch the COM client application that will work with your server.
- Continue profiling in the usual manner.

Profiling Out-of-Process COM Servers

Out-of-process COM servers are applications (not DLLs or OCXs) and they are executed in a separate address space. If you need to profile these types of programs, you need to load them in AQtime as a project, not as an additional module in another project. You can profile out-of-process servers either in Normal or in COM Server mode. To decide which profiling mode to choose, see *Profiling COM Applications*.

To profile your application in **Normal mode**, follow these steps:

- Compile your out-of-process COM server with debug information. See *How AQtime Profilers Use Metadata and Debug Information* for detailed information on how to do this.
- Be sure the "debug" version of your COM server is registered in the system. If the server was compiled on your machine, it was registered during compilation. In any case, you can use the regsvr32 utility located in the <Windows>\System folder to register the server, for example:

```
regsvr32.exe <Path>\MyServer.exe
```

- Open the out-of-process COM server in AQtime as a project and specify the profiling areas, triggers and actions (see *Controlling What to Profile*).
- Select **Normal** from the **Profiling Mode** dropdown list box that is displayed on AQtime's **Standard** toolbar (on the **AQtime** toolbar if you use AQtime integrated into Visual Studio; on the **Run.AQtime** toolbar if you use AQtime integrated into Borland Developer Studio).
- Start profiling. Make sure that the profiled application (that is, the server) can find all additional modules it requires.
- Launch the COM client. This is your "user" for the profiled application (the server).
- Work with the COM client and COM server application as needed.

© 2010 AutomatedQA Corp.

- Close the client and server applications. We recommended that you first close the COM client and then the COM server.

Note that since you start the COM server from AQtime, the latter always profiles the initialization and finalization code. If you need to profile this code when your COM server is launched by the operating system, use the **COM Server mode**:

- Compile your out-of-process COM server with debug information. See *How AQtime Profilers Use Metadata and Debug Information* for detailed information on how to do this.
- Be sure the "debug" version of your COM server is registered in the system.
- Open the out-of-process COM server in AQtime as a project and specify the profiling areas and triggers (see Controlling What to Profile).
- Select **COM Server** from the **Profiling Mode** dropdown list box that is displayed on AQtime's **Standard** toolbar (on the **AQtime** toolbar if you use AQtime integrated into Visual Studio; on the **Run.AQtime** toolbar if you use AQtime integrated into Borland Developer Studio).
- Select **Run | Parameters** from AQtime's main menu (select **Profile | Parameters** in Visual Studio or click **Parameters** on Borland Developer Studio's **Run.AQtime** toolbar). This will open the Run Parameters Dialog (for COM Server Mode). In the dialog:
  - In the **Client Application** box, specify the full path to your COM client application's executable.
  - Press **OK** to close the dialog.
- Press **Run** to start profiling (**Profile | Run** in Visual Studio or **Run | Run With Profiling** in Borland Developer Studio). AQtime will display a message informing that you should run the client application.
- Launch the COM client.
- Work with the COM client and COM server application as needed.
- Close the client and server applications.

Profiling DCOM Servers

DCOM applications are profiled in the same manner as out-of-process servers. Normally DCOM servers simply wait for a remote procedure call from a client machine. They cannot be launched by AQtime in this way. That is why the preferred profiling mode for them is the COM Server mode (this mode is supported on Windows NT, 2000, XP and Windows Server 2003):

- Compile your DCOM application with debug information. See *How AQtime Profilers Use Metadata and Debug Information* for detailed instructions on how to do this.
- Be sure the "debug" version of your application is registered in the system. If the application was compiled on your machine, it was registered during compilation. In any case, you can use the regsvr32 utility located in the <Windows>\System32 folder to register the control, for example:

```
regsvr32.exe <Path>\MyServer.DLL
```

- Open your application in AQtime.
- Select **COM Server** from the **Profiling Mode** dropdown list box that is displayed on AQtime's **Standard** toolbar (on the **AQtime** toolbar if you use AQtime integrated into Visual Studio; on the **Run.AQtime** toolbar if you use AQtime integrated into Borland Developer Studio).

- Select **Run | Parameters** from AQtime's main menu (select **Profile | Parameters** in Visual Studio or click **Parameters** on the **Run.AQtime** toolbar in Borland Developer Studio). This will open the Run Parameters Dialog (for COM Server Mode). In the dialog:
  - In the **Client Application** box, specify the application that will load your DCOM server. For example, this can be *<Windows>\System32\dllhost.exe*.
  - Press **OK** to close the dialog.
- Press **Run** to start profiling (**Profile | Run** in Visual Studio or **Run | Run With Profiling** in Borland Developer Studio). AQtime will display a dialog asking you to launch the client application.
- Launch the client application for your DCOM server.
- Perform profiling in the usual manner.

You can also profile DCOM servers in **Normal** mode. There is one DCOM server feature you should keep in mind: if these servers are launched as out-of-process COM servers in Normal mode, they will not execute anything (there is no remote procedure call) and will exit immediately. The solution is to add a code snippet to the DCOM server application so that, when launched, it does not close immediately. You can achieve this by:

- Adding an empty form (the Close button on the caption bar will allow the application to close when you are done).
- Setting up code so that when launched the DCOM application opens the form (the exact means depend on your compiler).

You can then profile your application using the rest of the recipe for out-of-process OLE servers. Use a client machine to command operations on the DCOM server. When you close the form on the server machine, the server process will exit and AQtime will generate its results.

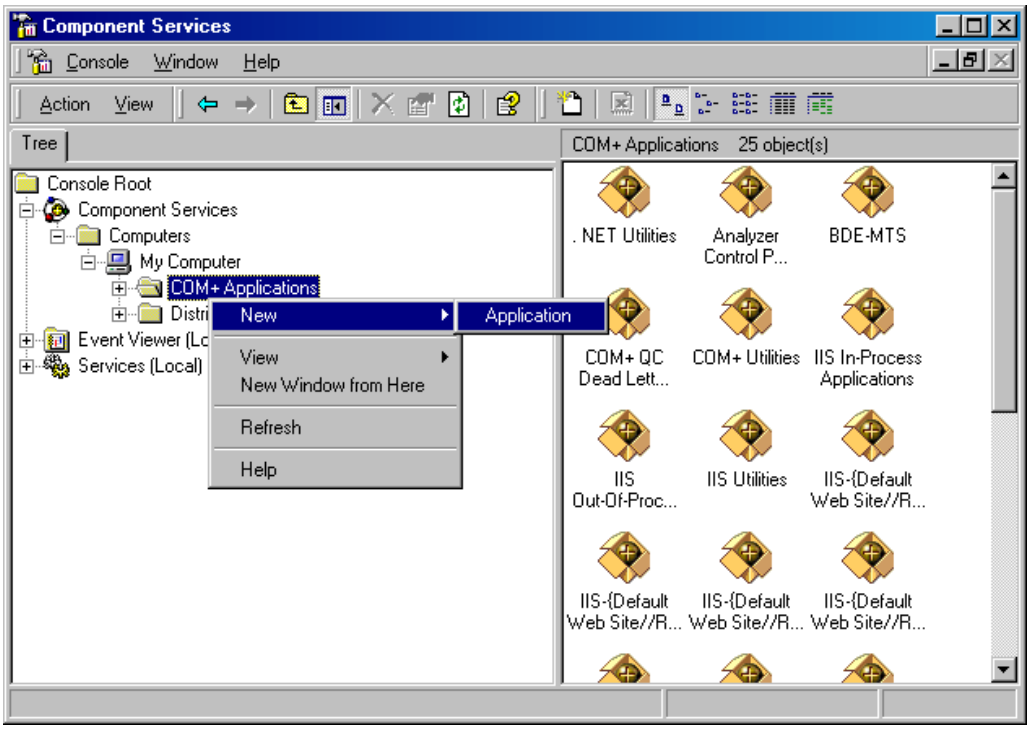
Note once again that the ways of profiling in Normal mode are just workarounds. In most cases, you should profile DCOM servers using AQtime's COM Server mode.

Profiling COM+ and MTS Applications

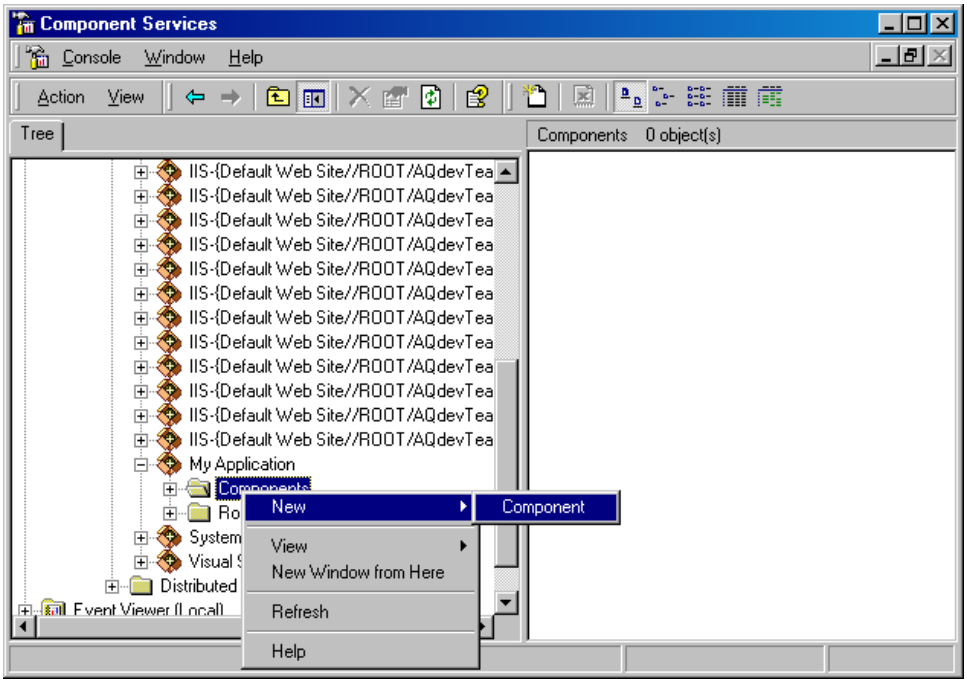
COM+ is the next generation of Microsoft's Component Object Model (COM). It is built on the basis of COM and MTS (Microsoft Transaction Server). This topic explains how to profile COM+ applications with AQtime.

- Compile your application with debug information. See *How AQtime Profilers Use Metadata and Debug Information* for detailed instructions on how to do this.
- Register your COM+ application in the operating system:
- Open the Component Services dialog (Control Panel | Administrative Tools | Component Services).
- Right-click the **COM+ Applications** item in the treelist on the left of the dialog. Select **New | Application** from the context menu. This will call the **COM Application Install Wizard**. Follow the steps of this wizard to create a COM+ application.

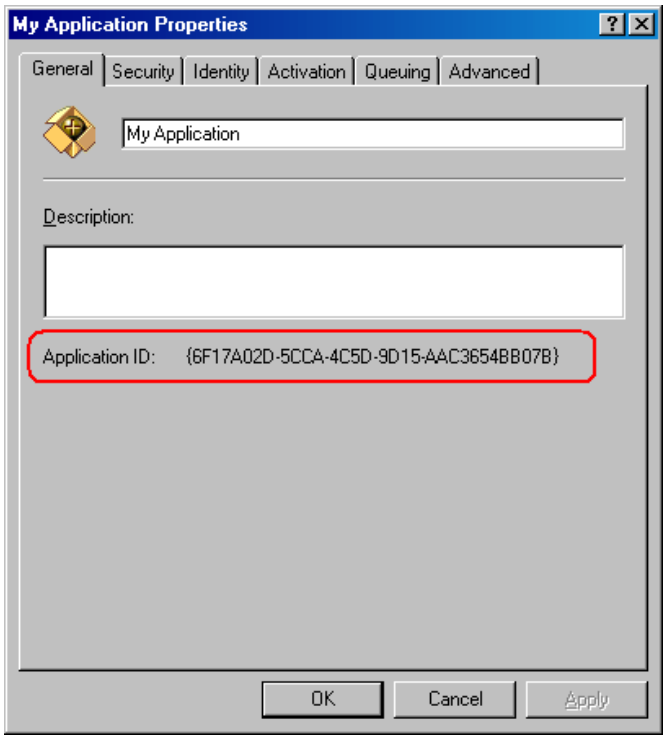
© 2010 AutomatedQA Corp.



- Once the application has been created, right-click the **<Your app> | Components** node and select **New | Component** from the context menu. This will call the **Component Install Wizard**. Use this wizard to add components to your application.



- Right-click the application node in the treelist and select **Properties** from the context menu. This will call the **Properties** dialog:



- Copy the **Application ID** field to the clipboard. You will need it for profiling.

You have registered the application in the operating system. Now, you may profile it with AQtime. You can profile your COM+ application like other COM applications in two modes: Normal and COM Server. For information on difference between them, see *Profiling COM Applications*.

To profile your application in **Normal mode**, follow these steps:

- Open your COM+ server in AQtime.
- Specify the profiling areas, triggers and actions (see Controlling What to Profile).
- Select the desired profiler.
- Select **Normal** from the **Profiling Mode** dropdown list box that is displayed on AQtime's **Standard** toolbar (on the **AQtime** toolbar if you use AQtime integrated into Visual Studio; on the **Run.AQtime** toolbar if you use AQtime integrated into Borland Developer Studio).
- Select **Run | Parameters** from AQtime's main menu (select **Profile | Parameters** in Visual Studio or click **Parameters** on Borland Developer Studio's **Run.AQtime** toolbar). This will call the Run Parameters Dialog (for Normal Mode).
- In the dialog:
  - Specify *dllhost.exe* as the **Host Application**. By default, this executable is located in the <Windows>\System32 folder.
  - Paste the Application ID value to the **Parameters** edit box (You copied this value from the Properties dialog of your COM+ application).
  - Press **OK** to close the dialog.
- Press **Run** to start profiling (**Profile | Run** in Visual Studio or **Run | Run With Profiling** in Borland Developer Studio). The dllhost.exe application will create the COM+ server with the ID specified in its command line.
- Run the client application and perform profiling in the usual manner.

To obtain profiling results, select **Run | Get Results** from AQtime's menu (select **Profile | Get Results** from the main menu of Visual Studio or click **Get Results** on the **Run.AQtime** toolbar in Borland Developer Studio).

To stop your COM+ server, either select **Terminate** from AQtime's Standard toolbar (select **Profile | Terminate** from the main menu of Visual Studio or click **Terminate** on the **Run.AQtime** toolbar in Borland Developer Studio) or right-click on the application node in Component Services and select **Shut Down** from the context menu. Note that Terminate and Shut Down call the Windows API `TerminateProcess` function. This function kills the process and does not send any notifications to AQtime. So, AQtime does not generate any profiling results. That is why you should select **Get Results** before closing the server.

To profile your application in **COM Server mode**, perform the following steps (note that COM Server mode is supported only on Windows NT, 2000, XP and Windows Server 2003):

- Select **COM Server** from the **Profiling Mode** dropdown list box that is displayed on AQtime's **Standard** toolbar (on Visual Studio's **AQtime** toolbar or on Borland Developer Studio's **Run.AQtime** toolbar).
- Select **Run | Parameters** from AQtime's main menu (select **Profile | Parameters** in Visual Studio or click **Parameters** on Borland Developer Studio's **Run.AQtime** toolbar). This will open the Run Parameters Dialog (for COM Server Mode). In the dialog:
  - In the **Client Application** box specify the name of the executable that will load COM+ server. This can be, for example, *dllhost.exe* located in the <*Windows*>*System32* folder.
  - Press **OK** to close the dialog.
- Press **Run** to start profiling (**Profile | Run** in Visual Studio or **Run | Run With Profiling** in Borland Developer Studio). AQtime will display a message box informing you that you should launch the client application.
- Run the client application and profile the application as you normally would.

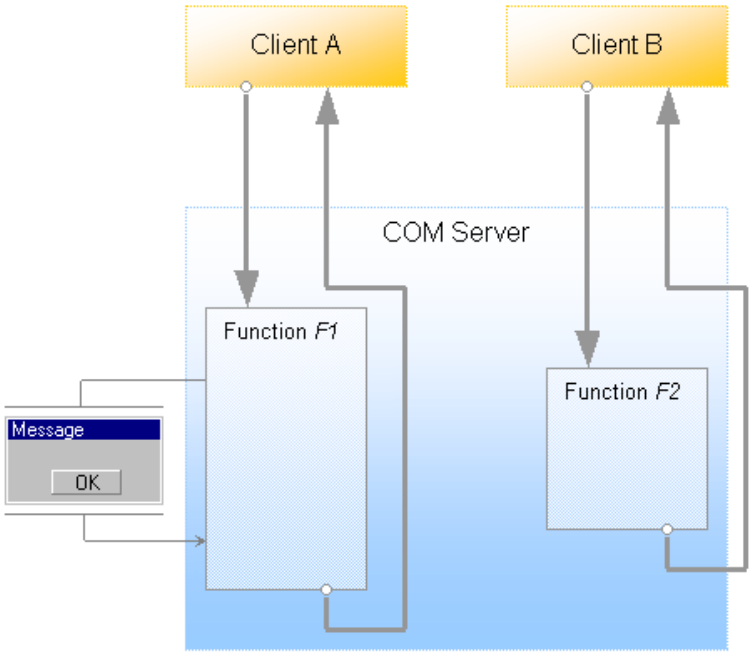
To obtain profiling results, select **Run | Get Results** from AQtime's menu (select **Profile | Get Results** from Visual Studio's menu or click **Get Results** on the **Run.AQtime** toolbar in Borland Developer Studio).

To stop your COM+ server, either select **Terminate** from AQtime's Standard toolbar (select **Profile | Terminate** from the main menu of Visual Studio or click **Terminate** on the **Run.AQtime** toolbar in Borland Developer Studio) or right-click on the application node in Component Services and select **Shut Down** from the context menu.

Profiling COM Logical Threads

A COM server may receive several requests from client applications simultaneously. To profile such a situation correctly, the Allocation, BDE SQL, Coverage, Function Trace, Performance, Reference Count and Resource profilers include a *Thread model* option that should be set to ***COM threads***. This topic explains why this is needed.

Suppose, you profile a COM server with two client applications (see the figure below):



Client A calls the server function `F1`. Suppose, this function executes for a long time (As shown on the figure above, the function is displaying a message box. So, it will execute until a user closes this message box). Now imagine that another client application, *Client B*, calls the server function `F2`, which finishes earlier than `F1` (because the message box has not been closed).

Both functions can be executed in the same Windows thread. "This is the default implementation of the COM servers with the Apartment threading model. COM servers with other threading models can also process several client requests within the same thread." you will probably say. The problem is that since `F2` starts and finishes during the `F1` execution and both functions are run within the same thread, the Performance profiler "thinks" that `F2` is a child function of`F1`. This leads to inaccurate profiling results: the execution time of the `F2` function is added to the time-with-children value of `F1`, the Details, Call Graph and Call Tree panels display `F2` as a child of `F1`, etc.

Setting the *Thread model* option to *COM threads* solves the problem. After you specify this value, AQtime will perform additional check to determine the "relationship" of the COM server functions. This helps avoid getting inaccurate profiler results. Note, however, that the additional check takes some time. If the profilers always did this check, the profiling speed would be slower. You should use the *COM threads* value if you profile a COM server with several clients, or if several threads in the same client application include server function calls. The COM server threading model (apartment, single, etc.) is of no importance. The only factor that specifies whether you should use this value is the number or clients.

Profiling Scripts

Code written in a scripting language is similar to code written in a traditional programming language. So, it may have performance issues too. This topic describes how you can profile script code with AQtime.

- *Supported Scripting Languages and Engines*
- *Preparation*
- *Supported Profilers*
- *Profiling Steps*

© 2010 AutomatedQA Corp.

- *Analyzing Script Profiling Results*
- *Troubleshooting*

Supported Scripting Languages and Engines

You can use AQtime to profile scripts executed by the Microsoft Scripting Engine. For example, this engine is used by Internet Explorer, TestComplete and Windows Script Host (this utility allows you to run scripts directly from Microsoft Windows). The engine supports the VBScript and JScript scripting languages. Thus, AQtime can profile web scripts, TestComplete projects and Windows Script Host tasks that are written in these two languages.

You can also profile TestComplete projects created with C#Script and C++Script , as in TestComplete, C#Script and C++Script are based on JScript.

Currently, AQtime cannot profile TestComplete's DelphiScript projects and web scripts from pages displayed in the Firefox and Opera browsers, because neither of these browsers, nor this language uses the Microsoft Scripting Engine.

AQtime cannot profile server-side scripts of ASP pages. It can only profile client scripts of ASP pages that are run by the browser.

AQtime supports profiling scripts that are run by 64-bit versions of Internet Explorer, but it cannot profile scripts executed in other 64-bit processes. This happens because there are no script debugging libraries that can be used in these processes.

Preparation

Script code is executed by an application that hosts the script engine. In order for AQtime to be able to collect scripting data, you need to configure your computer and host application before profiling. For complete information on the actions to be performed, see *Profiling Scripts - Prerequisites*.

Supported Profilers

Currently, scripts can be profiled with the following profilers:

- Performance Profiler
- Coverage Profiler
- Light Coverage Profiler
- Function Trace Profiler

Profiling Steps

To profile a script, you should launch the host application under AQtime and then execute the script in that application. AQtime recognizes the script's activity and reports its results along with the application results. There are three ways to profile a script, they vary in the module that is added to the project: a host application, a script file or a URL. See the following topics for complete information:

- *Profiling Scripts Using Host Applications*
- *Profiling Script Files*
- *Profiling Scripts Located on Web Pages*

If you profile scripts running in Internet Explorer 8, then before you start profiling, close all the Internet Explorer windows. Do not open windows or pages (except for the profiled window and page) until the profiling is over.

Read the *Profiling Scripts in Internet Explorer 8* topic to learn about the other peculiarities of

profiling under this browser.

Analyzing Script Profiling Results

The results of script profiling are reported along with the host application's profiling results. There is no significant difference between results produced upon profiling ordinary applications and libraries and results of script profiling.

To distinguish the results of script profiling from the results of host application profiling, you can use the Code Type column of the Report panel. The values of this column indicate the type of the routine's code. For script results this column holds the *Script* value. You can group or filter the results by this value. By default, the Code Type column is hidden, but you can display it via the **Field Chooser**. See *Adding and Removing Columns*.

Troubleshooting

For information on how to resolve problems that may occur during script profiling, see *Profiling Scripts - Troubleshooting*.

Profiling Scripts – Prerequisites

This topic describes the actions you may need to perform in order to profile scripts with AQtime. These actions include:

1. Checking for required components
2. Setting user permissions
3. Preparing the host application for script profiling
   - **3.1** Preparing Internet Explorer
   - **3.2** Preparing TestComplete
   - **3.3** Preparing Windows Script Host

1. Checking for Required Components

The script profiling feature requires the Windows Script and Script Debugger components. The Windows Script component is supplied with each Microsoft operating system (since Windows 98), thus it is already present on your machine. The Script Debugger may also be installed on your machine, as it is shipped along with Visual Studio (since version 2003) and Microsoft Office (version XP and later).

If you have problems with profiling scripts, try reinstalling these components. You can find the standalone packages at **Microsoft Download Center** (http://www.microsoft.com/downloads). Just search for the following items:

- Windows Script (Windows Script 5.6 or later is required)
- Script Debugger

The versions of Windows Script and Script Debugger may be different and incompatible with one another. For information on possible problems and workaround for them, see *Profiling Scripts – Troubleshooting*.

2. Setting User Permissions

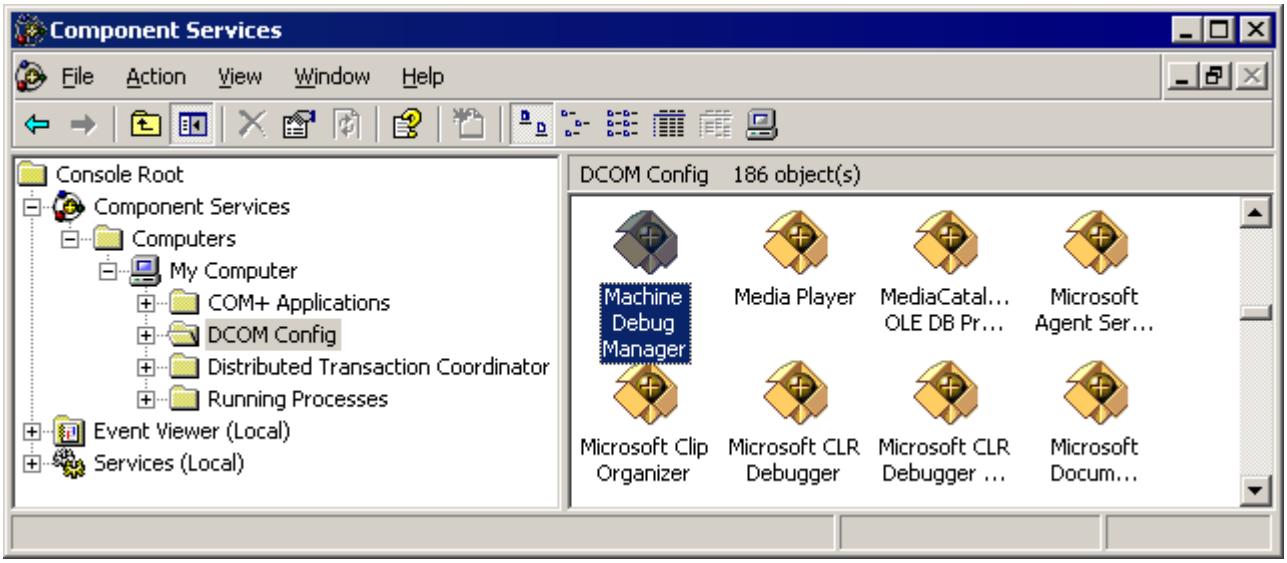
In order to debug script code, your user account must have the *Launch and Activation* and *Access* permissions for the Machine Debug Manager system component. By default, these permissions are granted to user accounts that belong to the *Administrators* and *Debugger Users* groups. To obtain the permissions, ask

your administrator to add your account to these groups. An alternative way is to ask your administrator to modify permissions for the Machine Debug Manager component.

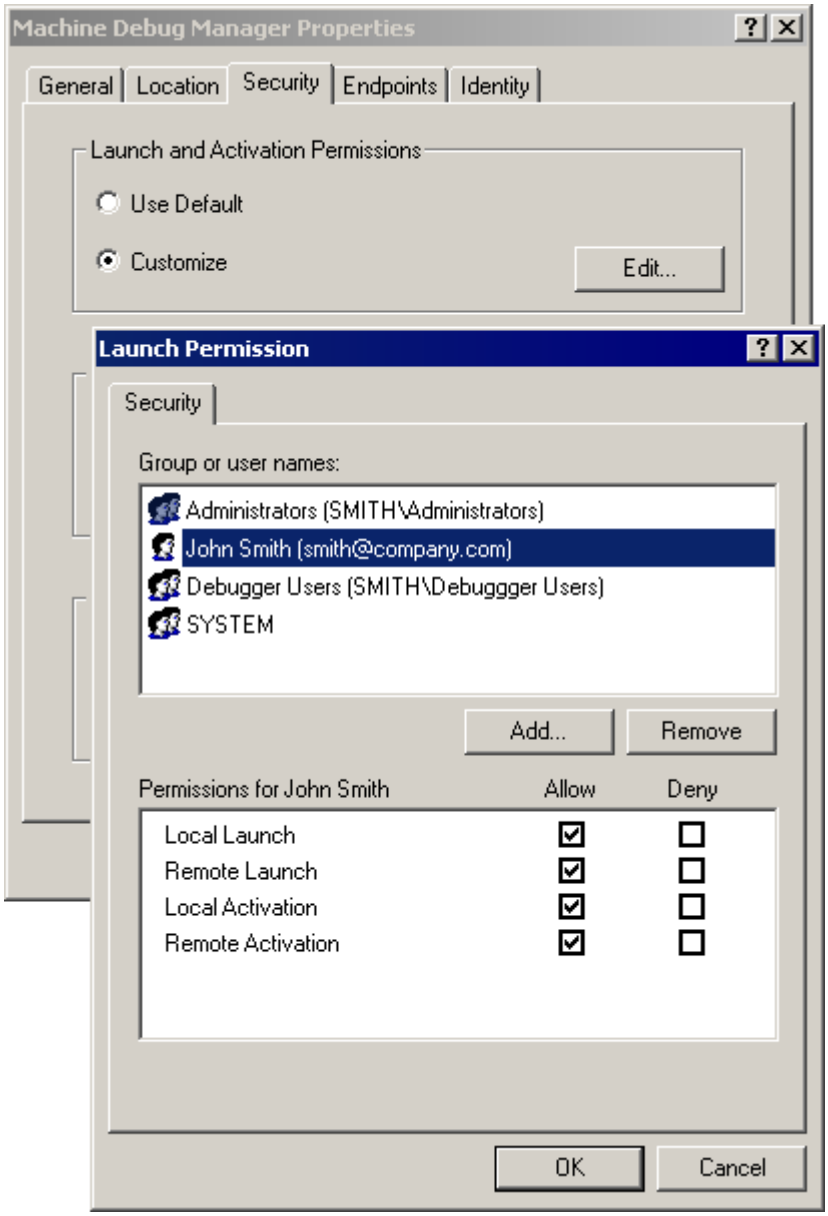
You can change the permissions using the Component Services snap-in:

- The way you open the snap-in depends on your operating system:
  - In Windows XP and earlier versions - open the **Control Panel | Administrative Tools | Component Services** from the Start menu.
  - In Windows Vista and latter versions - add the Component Services snap-in to the Microsoft Management Console (MMC):
    - Type “mmc” in a command window and press ENTER. The Microsoft Management Console will appear.
    - Select the **File | Add/Remove Snap-in...** from the main menu.
    - Press Add in the Add Standalone snap-in dialog.
    - Select the **Component Services** from the list and click **Add** to confirm adding.

  Now the Component Services snap-in is available for configuring.
- Select the **Console Root | Component Services | Computers | My Computer | DCOM Config** node in the tree on the left of the window.



- Right-click **Machine Debug Manager** in the list of available components and choose Properties from the context menu.
- Switch to the **Security** tabbed page of the subsequent dialog.
- Choose **Custom** in the **Launch and Activation Permissions** group and press **Edit**.
- Select your user account in the ensuing dialog, enable the Allow checkmark at least for the **Local Launch** and **Local Activation** items and press **OK** to close the dialog.



- Choose **Custom** in the **Access Permissions** group and press **Edit**.
- Select your user account in the ensuing dialog, enable the Allow checkmark at least for the **Local Access** item and press **OK** to close the dialog.
- Press **OK** in the Machine Debug Manager Properties dialog to close it and save the changes.

3. Preparing the Host Application for Script Profiling

AQtime gathers data about scripting activity from the Microsoft Scripting Engine. It tracks the debug information that is passed to the engine. Therefore, in order to obtain profiling results in AQtime, the following conditions should be met:

a. The host application should generate debug information

b. The debug information should be handled by the Microsoft Scripting Engine

© 2010 AutomatedQA Corp.

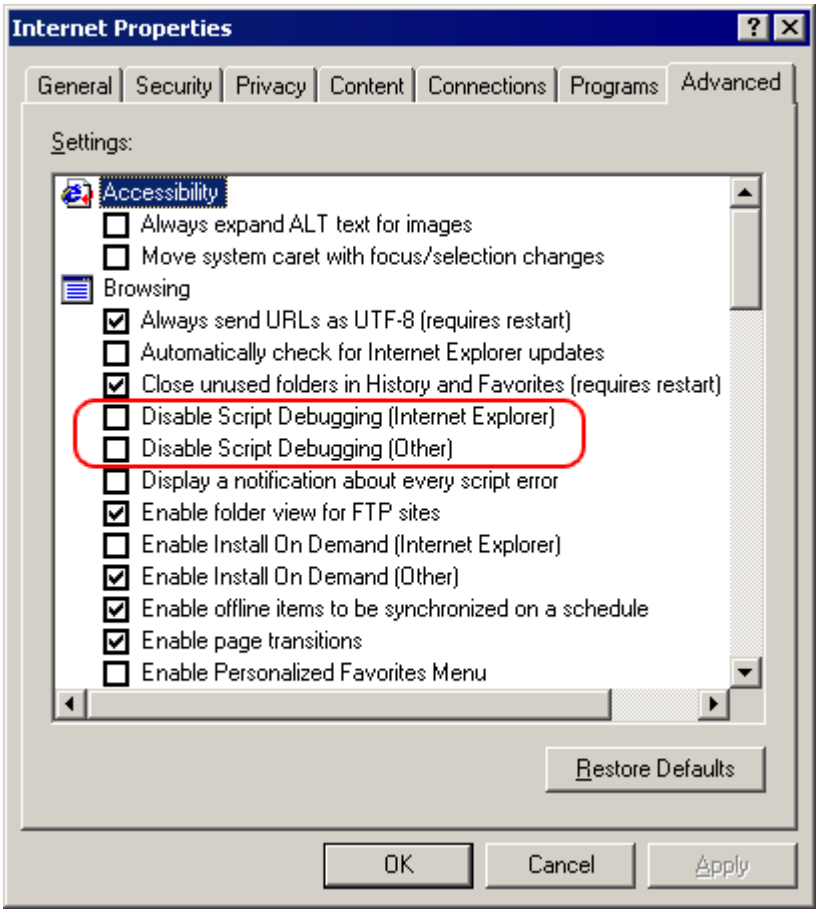
The settings that allow meeting these conditions depend on a particular host application. The settings for frequently used script hosts are described below.

3.1 Preparing Internet Explorer

To intercept the scripting activity from Internet Explorer ver. 7 and earlier, you should enable the browser's script debugger. If you use Internet Explorer 8, no script debugger is required. See *Profiling Scripts in Internet Explorer 8*.

To enable the browser's script debugger:

- Open the Control panel.
- Select **Internet Options**.
- Switch to the **Advanced** page of the ensuing **Internet Options** dialog.
- Clear the **Disable Script Debugging (Internet Explorer)** and **Disable Script Debugging (Other)** check boxes in the **Browsing** category.



- Click **OK** to save the changes and close the dialog.

3.2 Preparing TestComplete

To get the results of profiling TestComplete scripts, you should ***disable*** the in-process debugger of TestComplete. This should be done because by default, TestComplete's native debugger is set to handle scripts by itself, without passing data to the system.

This behavior can be changed in TestComplete by releasing the **Enable Script Debugging** button on the **Debug** toolbar, or by disabling the **Script | Enable Script Debugging** item of the main menu.

3.3 Preparing Windows Script Host

Windows Script Host allows you to run scripts directly from Microsoft Windows. It is represented by two executables: WScript and CScript. They operate in a similar way; the only difference is that WScript generates windowed output, while CScript sends its output to a console. The scripts to be launched, as well as host options and script parameters, are passed to Windows Script Host via command-line arguments.

To obtain profiling results from Windows Script Host, you should enable its built-in debugger and send the script's output to the console. This behavior is controlled by the `//D` and `//H:CScript` host options respectively. That is, to get profiling results in AQtime, you should use the following syntax: `WScript.exe` or `CScript.exe //D //H:CScript Other_Host_Options "Path_To_Script" Script_Parameters`. For example, `WScript.exe //D //H:CScript "C:\Work\MyScript.js"`.

Profiling Scripts Using Host Applications

There are three ways to profile script code with AQtime. They vary in the module that is added to the project: a host application, a script file or a URL. The technique that uses host applications is the simplest. It does not require that additional parameters be specified for the profiler. When profiling starts, the host application is launched and you can load any script you want. However, the weak point of this technique is that you cannot select the script routines to be profiled via the Setup panel. That is, AQtime will profile all routines that were executed. This technique cannot be used for Windows Script Host, since it has no user interface to open script files.

To profile a script via its host application, do the following:

- Add the host application's executable (iexplore.exe for Internet Explorer; testcomplete.exe for TestComplete) to the Modules pane.

  This can be done by selecting **Add Module** from the **Setup** toolbar or context menu, or by choosing the **Project | Add module** menu item.
- Select the desired profiler. In Standalone version use the dropdown list on the Standard toolbar, in integrated versions use the **Profiler** dropdown list in the **Profile** menu of Visual Studio or the **Current Profiler** submenu of the **Profile** menu of Borland Developer Studio.
- Start profiling. That is, press **Run** on the Standard toolbar or select **Run | Run** from AQtime's main menu, select **Profile | Run** from Visual Studio's main menu, or select **Run | Run With Profiling** from Borland Developer Studio's main menu.
- AQtime will launch the chosen host application. In the application, open the script and call the routines you want to profile. That is, open and run the desired suite, project, unit or routine in TestComplete, or open the desired web page in the browser and do the actions needed to run the desired routine (click on links, buttons, menus and so on).
- Wait until these routines are executed and close the host application. AQtime will generate profiling results and display them in the Report and Summary panels.

Profiling Script Files

There are three ways to profile script code with AQtime. They vary in the module that is added to the project: a host application, a script file or a URL. To profile a script file, add it to the list of profiled modules in AQtime's Setup panel. AQtime considers a file as a script file if it contains VBScript, JScript, C#Script or

C++Script code, or if it contains HTML code in which <script> tags are used. Both the ASCII and Unicode encodings are possible. Typically, files that contain VBScript and JScript code have the *.vbs* and *.js* extensions, while TestComplete scripts have the *.svb*, *.sj*, *.scs* and *.scpp* extensions (for VBScript, JScript, C#Script and C++Script respectively).

Script files cannot be executed by themselves, they require that the host application be specified. For *.vbs* and *.js* files, the host application can be either Internet Explorer or Windows Script Host; for *.htm* and *.html* files the host application should be Internet Explorer; for *.svb*, *.sj*, *.scs* and *.scpp* files the host application should be TestComplete.

This approach has another benefit: when a file is added to the Modules pane, AQtime retrieves data on available routines. Thus, you can select which script routines to profile in the same manner as for ordinary libraries and executables. See *Defining Areas to Profile* topic.

To profile a script file, follow the instructions below:

- Press the **Add Module** button on the Modules pane, or select **Add Module** from the pane's context menu, or choose the **Project | Add module** item of the main menu.
- In the ensuing **Add Module To Project** dialog, select the *All Files* or *Script Files* file mask and locate the desired script file. Press **OK** to add it to the AQtime project.
- Optional. Choose the elements to be profiled by adding scripting routines to including or excluding profiling areas.
- Ensure that **Normal** is currently chosen in the Profiling Mode dropdown list box and select
  - **Run | Parameters** from the AQtime's main menu,
  - **Profile | Parameters** from Visual Studio's menu
  - **Profile | Parameters** from Developer Studio's menu

  to call the Run Parameters dialog.
- In the dialog, specify the host application and parameters for it. Depending on the script file type, do one of the following:

  Web Script Files (*.vbs, *.js, *.htm, *.html, *.asp, *.php)

  - Enter the path to iexplore.exe in the **Host Application** field. Alternatively, if Internet Explorer is your default browser, you can select **Default Web Browser** from the editor's drop-down list.
  - Specify the URL or the path to the web script file in the **Parameters** field. This will make the browser navigate to the specified page upon starting.

  TestComplete Script Files (*.svb, .sj, .scs* and *.scpp*)

  - Enter the path to testcomplete.exe in the **Host Application** field.
  - Specify the fully qualified path to the project or project suite in the **Parameters** field. This will make TestComplete open the specified project or project suite.

  Standalone Script Files (*.vbs, *.js*)

  - Enter the path to any Windows Script Host executable, *wscript.exe* or *cscript.exe*, in the **Host Application** field.
  - In the **Parameters** field, specify `//D //H:CScript` followed by the fully qualified path to the file and script parameters (if any). This will enable the host's built-in debugger and make the host application launch the specified script upon starting profiling in AQtime.
- Close the dialog by pressing **OK**.

- Select the desired profiler. In Standalone version use the dropdown list on the Standard toolbar, in integrated versions use the **Profiler** dropdown list in the **Profile** menu of Visual Studio or the **Current Profiler** submenu of the **Profile** menu of Borland Developer Studio.
- Start profiling. That is, press **Run** on the Standard toolbar or select **Run | Run** from AQtime's main menu, select **Profile | Run** from Visual Studio's main menu, or select **Run | Run With Profiling** from Borland Developer Studio's main menu. The host application will be launched automatically.
- Start the script routine(s) to be profiled. Depending on the script file type, various actions should be performed:

  Web Script Files (*.vbs, *.js, *.htm, *.html, *.asp, *.php)

  Typically, web page scripts are executed upon opening. However, sometimes, specific actions may be needed, for example, you may need to highlight elements, select menus, interact with the page's controls and so on.

  TestComplete Script Files (*.svb, .sj, .scs* and *.scpp*)

  Run the desired routine, unit, project or project suite in TestComplete.

  Standalone Script Files (*.vbs, *.js*)

  No additional actions are required as Windows Script Host automatically starts the script file specified as a parameter.
- Wait for the routine(s) to complete and close the host application. AQtime will generate the profiling results and display them in the Report and Summary panels.

Profiling Scripts Located on Web Pages

There are three ways to profile script code with AQtime. They vary in the module that is added to the project: a host application, a script file or a URL (web page). Using a web page is a better way of adding a script file to your project. It allows you to profile scripts right on web pages. A page can be on the Internet, on a local network or on a local computer. If a remote URL is specified, the corresponding page is copied to a temporary folder on your computer. AQtime retrieves information from HTML, ASP, PHP and other pages that contain <script> tags. If some script snippet does not define a routine name, then the results will be reported as *<Global Scope of [URL]>*.

To profile a web script, follow the instructions below:

- Press the **Add URL** button on the Modules pane, or select **Add URL** from the pane's context menu. This will call the Add URL to Project dialog.
- Specify the desired web page in the dialog and press **OK**.
- Once the page is added, AQtime retrieves information about available script routines. Thus, you can choose the elements to be profiled.
  - Ensure that **Normal** is currently chosen in the Profiling Mode dropdown list box and select
    - **Run | Parameters** from the AQtime's main menu,
    - **Profile | Parameters** from Visual Studio's menu
    - **Profile | Parameters** from Developer Studio's menu

    to call the Run Parameters dialog.
- In the dialog, specify the executable of Internet Explorer as the host application. You can enter the executable's path in the **Host Application** editor or select **Default Web Browser** from the editor's drop-down list (if Internet Explorer is your default browser). Press **OK** to close the dialog.

- Select the desired profiler. In Standalone version use the dropdown list on the Standard toolbar, in integrated versions use the **Profiler** dropdown list in the **Profile** menu of Visual Studio or the **Current Profiler** submenu of the **Profile** menu of Borland Developer Studio.
- Start profiling. That is, press **Run** on the Standard toolbar or select **Run | Run** from AQtime's main menu, select **Profile | Run** from Visual Studio's main menu, or select **Run | Run With Profiling** from Borland Developer Studio's main menu. The browser will be launched automatically.
- In the browser, enter the page's URL in the address bar to navigate to the page. Typically, web page scripts are executed upon opening. However, sometimes, to call the desired routine you may need to perform some specific actions: highlight elements, select menus, interact with the page's controls and so on.
- Wait for the routine(s) to finish and close the browser. AQtime will generate the profiling results and display them in the Report and Summary panels.

Profiling Scripts in Internet Explorer 8

The eighth version of the Microsoft Internet Explorer browser introduced a number of enhancements. It significantly differs from its predecessors. The internal structure of the browser has also been modified (for example, a separate iexplore.exe process is created for each new tab). To reflect these changes, AQtime 6.30 introduces a special mode of script profiling that is optimized for Internet Explorer 8. This mode allows you to obtain profiling results faster and does not require the script debugger to be enabled.

The mode activates automatically when AQtime determines that the host process is Internet Explorer 8. This mode is available for all three ways of script profiling (via the host application, via a script file or via a URL).

> **Note:** In order to obtain the correct results when profiling in Internet Explorer 8 mode, close all Internet Explorer windows before you start profiling. Do not open pages and windows until profiling is over.

Internet Explorer 8 mode has the following peculiarities and restrictions:

- **Routine level profiling** - Profiling is performed at the routine level only, line level profiling is not supported. Yet, since the version of the host process cannot be determined prior to launching it, the Line Level radio button is still enabled in the Areas pane.
- **Verbose results** - Profiling results include data for routines contained on web pages or in module files, data for object method calls, and data for helper service routines of other browsers.
- **Extended data structure names** - Routines and other data structures are named according to the internal notation of Internet Explorer. So, in profiling results, you may encounter items like "`Main, JScript - window script block`", and other items that have extended descriptions. If a routine does not have a custom name (many helper routines have no name), then it is entitled using the following pattern: "`Routine Id: rID; Script Id: sID; Routine name: Unknown`", where *`rID`* and *`sID`* stand for routine and script identifiers, respectively.
- **JScript only profiling** - The engine of Internet Explorer can profile scripts written in JScript. Scripts written in VBScript are exececuted by the engine, but the profiling results cannot be obtained.

Profiling Scripts – Troubleshooting

There are a number of requirements when profiling scripts. If you get empty results when profiling scripts or if you cannot start profiling, most likely your computer or actions do not meet these requirements. Below are some tips that may help you solve the problem:

- First, check the profiler you use. Script profiling is only supported by the Performance, Coverage, Light Coverage and Function Trace profilers (currently, the Function Trace profiler does not collect information on values of script routines' parameters).
- Check if you are running multiple instances of AQtime. Only the first instance of AQtime can profile scripts and obtain results.
- Check whether the tested application uses the Microsoft Scripting Engine, since AQtime can only profile those scripts that are executed by this engine.

  Note that AQtime cannot profile server-side scripts of ASP pages. It can profile only client scripts of ASP pages that are run by the browser.

  Also, AQtime cannot profile scripts executed in 64-bit processes. This happens because there are no script debugging libraries that can be used in these processes. However, there is an exception: AQtime can profile scripts that are run by 64-bit versions of Internet Explorer.
- If you profile with the "Attach-to-Process" feature, change the approach and try launching the host application from AQtime (you can specify the host application for your script units in the Run Parameters dialog).
- Make sure that the Microsoft Script Debugger is installed on your computer. Note that there are several versions of the debugger and AQtime may not support some of them. For instance, there is a known problem with the script debugger installed by Microsoft Visual Studio 2008. Currently, AQtime cannot use this one for script profiling since the Visual Studio installation program replaces some debugger modules and AQtime stops working with the debugger properly. To avoid the problem, re-install Microsoft Script Debugger after installing Visual Studio 2008.

  To be sure you have the correct version of the debugger, follow these steps:

  - Download and install all the updates available for the operating system and Internet Explorer.

    There is one exception: While updating, you may be suggested to install Internet Explorer version 8. This update is not required, and you may skip it if you want to use another version of the browser.
  - Download and install Microsoft Script Debugger from this site: http://www.microsoft.com/downloads/details.aspx?FamilyId=2F465BE0-94FD-4569-B3C4-DFFDF19CCD99&displaylang=en.
  - Download the PDM.dll from this page: http://www.dll-files.com/dllindex/dll-files.shtml?pdm. This DLL implements some debugger's functionality.
  - Replace the PDM.dll file in the *<Windows>\System32* folder with the downloaded module.
  - On Windows Vista and later operating systems: temporarily disable User Account Control (UAC):
    - Open the Control Panel by clicking **Start** and then **Control Panel**.
    - In the search box, type *User Accounts* and select the **User Accounts** applet.
    - In the User Accounts window, click **Turn User Account Control on or off**.

The operating system will display a dialog box asking for your permission to continue. Click **Continue** in the dialog.

- Clear the **Use User Account Control (UAC) to help protect your computer** check box and click **OK**.
- Close the User Accounts window.
- Restart your computer.

- Register the downloaded module with the following command:

```
regsvr32.exe C:\Windows\System32\pdm.dll
```

- Restart your computer, if the changes do not immediately take effect.
- On Windows Vista and later operating systems: enable User Account Control.

- If you use Internet Explorer 8, then close all Internet Explorer windows before you start profiling. Do not open pages and windows until the profiling is over.

If these tips do not help, send a message to AutomatedQA Support Team (for information on how to do this, see *Getting Support*).

Profiling Services

Windows NT, Windows 2000, Windows XP and Windows Server 2003 support an application type called *service application*. To profile your service application with AQtime, perform the following operations:

1. Compile your service application in your development tool, for example, Microsoft Visual Studio or Borland Delphi. See *How AQtime Profilers Use Metadata and Debug Information* to decide whether or not to include debug information.
2. Register your service application as a service. Your service must be displayed in the Service Control Manager dialog (**Control Panel | Services** in Windows NT or **Control Panel | Administrator Tools | Services** in Windows 2000, XP or Windows Server 2003).
3. Create the AQtime project from the executable module of your service (**File | New Project From Module**).
4. Select **Service** from the **Profiling Mode** dropdown list box that is displayed on AQtime's **Standard** toolbar (on the **AQtime** toolbar if you use AQtime integrated into Visual Studio; on the **Run.AQtime** toolbar if you use AQtime integrated into Borland Developer Studio).
5. In AQtime running as a standalone application, select the desired profiler and click **Run** to start profiling. In AQtime integrated into Visual Studio, select the desired profiler in the **Profile | Profiler** menu item or from the **Profilers** box on the **AQtime** toolbar; then choose **Profile | Run** to start profiling. In AQtime integrated into Borland Developer Studio, select the desired profiler in the **Profile | Current Profiler** menu item; then choose **Run | Run With Profiling** to start profiling.

   AQtime will start your service (if the service is running, AQtime will restart it, so the service will be running in the debug mode under AQtime). AQtime will post notification messages to the Event View panel upon beginning and finishing the service start.

   Note: If your executable registers several services, only the first one is started by AQtime during profiling, so, AQtime will not be able to profile the other services. As a workaround, you can start profiling (the first service will be started), start the other services manually and then profile the application.
6. Test your service application according to your needs.

7. When your tests are complete, select **Run | Get Results** from AQtime's main menu (select **Profile | Get Results** from the main menu of Visual Studio or click **Get Results** on the **Run.AQtime** toolbar in Borland Developer Studio) to obtain profiling results. Another way to obtain profiling results is to create an action that will tell AQtime to generate the results. See *Getting Results During Testing* for more information. The results are also generated if stop the service via the Service Control Manager dialog. If you do not want to stop the service you can use the Get Results menu item or an action.

Profiling SQL Server CLR Integration Assemblies

Microsoft SQL Server 2005 hosts .NET Framework and provides developers with the possibility to create stored procedures, triggers and functions in C# .NET, Visual Basic .NET and Visual C++ .NET. The created assemblies are uploaded to SQL Server and stored in system catalogs. You can then create special database objects such as functions, procedures, triggers, types and aggregates, that will connect to the assemblies and call the CLR routines. For instance, you can create a Transact-SQL query that will call a routine from a CLR integration assembly the same way it calls other Transact-SQL functions.

For detailed information on creating CLR integration assemblies for Microsoft SQL Server, see the *Database Engine .NET Framework Programming* section of the MSDN Library.

Profiling of CLR integration assemblies with AQtime has a number of peculiarities:

- First of all, you should launch SQL Server from AQtime. The CLR integration assemblies are loaded in memory by the SQL Server process, and you cannot use the "attach to process" feature to connect the profiler to this process, because this feature does not support profiling of managed code. So, you have to start SQL Server from AQtime.
- To launch SQL Server, you should use certain run parameters in your AQtime project.
- Another important point is to select the *Profile Entire .NET Code* area in the Setup panel.

  The problem is that AQtime is unable to determine the module name of your integration assembly when this assembly is loaded by SQL Server. This happens due to certain peculiarities of SQL Server. Since the module name cannot be determined, the profiling area settings will not function. So, if you do not check the *Profile Entire .NET Code* box, AQtime will be unable to find the routines for profiling and you will get empty results.

  Since the module name cannot be determined, the triggers and actions that contain classes and routines defined in your assembly will also be ineffective. So, do not forget to set the *Initial Profiling Status* setting to ON in the Setup panel. If it is off, the triggers and actions will not be active during the run.
- In order for AQtime to be able to trace the execution of your assembly's functions, you should change the security settings of your database. To do this, you should execute specific SQL code (see below).
- Finally, to obtain profiling results, you should use the **Get Results** command (see *Getting Results During Testing*).

Below is a step-by-step description of how to profile SQL Server CLR integration assemblies.

Requirements

The CLR integration assemblies operate on the SQL Server computer. So, to profile them, you should install AQtime on this computer.

Upload your assembly to the server and register it in system catalogs. For more information on how to perform these actions, see the MSDN Library.

Before profiling your CLR integration assembly, test it in SQL Server Management Studio and ensure that the assembly functions as expected when it is not being profiled by AQtime.

1. Changing Database Security Settings

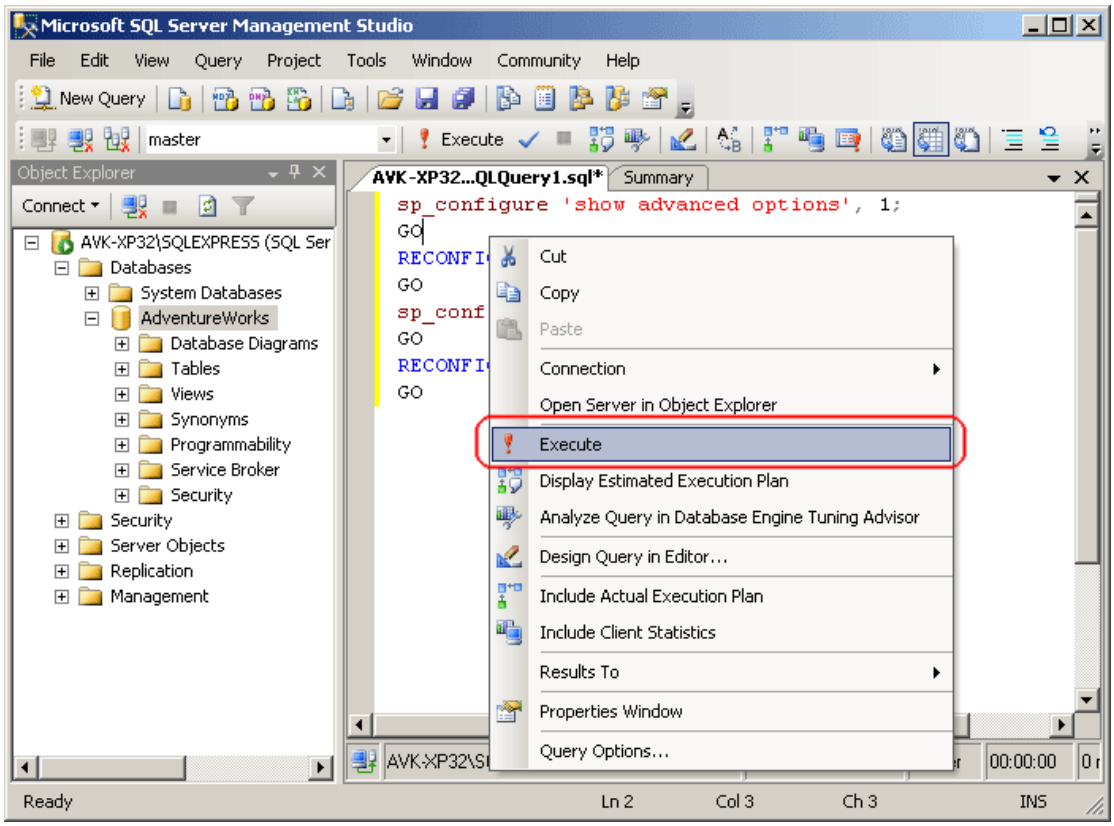
To modify the security settings, we will create and use a temporary query:

- Open **SQL Server Management Studio** and connect to the SQL Server instance that controls your database.
- Create a new empty query. To do this, choose **File | New | Query with Current Connection** from the Management Studio's main menu.
- Type the following code into the query editor:

```
[Transact-SQL]

sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
sp_configure 'clr enabled', 1;
GO
RECONFIGURE;
GO
```

- Execute the query. You can do this by right-clicking somewhere within the query editor and selecting **Execute** from the context menu.



- Clear the query code and type the following lines:

```
[Transact-SQL]

ALTER DATABASE Database_Name SET TRUSTWORTHY ON;
GO
```

Here, *Database_Name* stands for the name of your database. For instance,

```
[Transact-SQL]

ALTER DATABASE AdventureWorks SET TRUSTWORTHY ON;

GO
```

- Choose **Execute** from the context menu to execute this query.
- Clear the query code once again and type the following text:

```
[Transact-SQL]

USE Database_Name
ALTER ASSEMBLY Assembly_Name WITH permission_set = UnSafe
GO
```

Here, *Database_Name* stands for the name of your database and *Assembly_Name* is the file name of your assembly that implements the stored procedure functionality. You should specify the file name without the path and extension:

```
[Transact-SQL]

USE AdventureWorks
ALTER ASSEMBLY HelloWorld WITH permission_set = UnSafe
GO
```

- Execute this code.
- Close the query editor. Answer No when the SQL Server Management Studio asks you to save the changes to the query.

2. Setting AQtime Project

- Launch AQtime and add the desired assembly to your AQtime project.
- Select the **Profile Entire .NET Code** check box in the Setup panel.

  Note: It is important that you select *Profile Entire .NET Code*. Due to certain peculiarities of SQL Server, AQtime is unable to determine the module name of your SQL Server extension and the profiling area settings will not work. If you do not check the *Profile Entire .NET Code* box, you will get empty results.

- In the **Triggers and Actions** section of the Setup panel, set the **Initial Profiling Status for All Threads** option to ON.

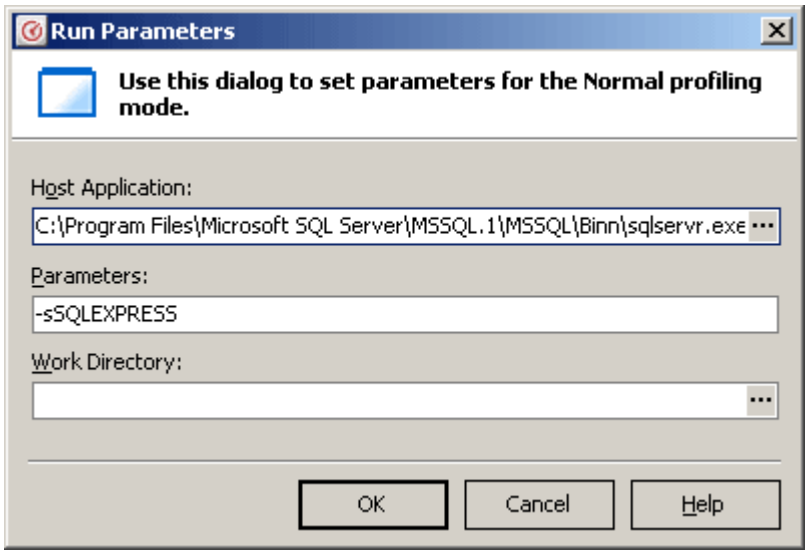
  Note: The requirement to enable the initial profiling status is caused by the fact that it is unable to determine the module name. When the module name cannot be determined, AQtime is unable to recognize the executed routines properly. So, the triggers and actions that contain the routines defined in your integration assembly, will not work. They will not turn the profiling on, if the initial profiling status is off.

Now we have to specify the run mode and run parameters:

- We will run the profiler in Normal mode. Select this mode from AQtime's **Standard** toolbar. (If you use AQtime integrated into Microsoft Visual Stuio, you can select this mode from Visual Studio's **AQtime** toolbar. If you use AQtime integrated into Borland Developer Studio, select the mode from Borland Developer Studio's **Run.AQtime** toolbar).
- Choose **Run | Parameters** from AQtime's main menu (if you use AQtime integrated into Microsoft Visual Studio, choose **Profile | Parameters** from Visual Studio's main menu. In Borland Developer Studio, choose **Profile | Parameters** from the main menu).

  This will invoke the **Run Parameters Dialog (for Normal Mode)**. Specify the following values in this dialog:

  - In the **Host Application** box specify the path to the *<Program Files>/Microsoft SQL Server/MSSQL/Binn/sqlservr.exe* module.
  - In the **Parameters** box specify the *-s* command-line argument followed by the server name. Typically, you should type **-sMSSQLSERVER**, if you use Microsoft SQL Server 2005, or **-sSQLEXPRESS**, if you use Microsoft SQL Server Express Edition.



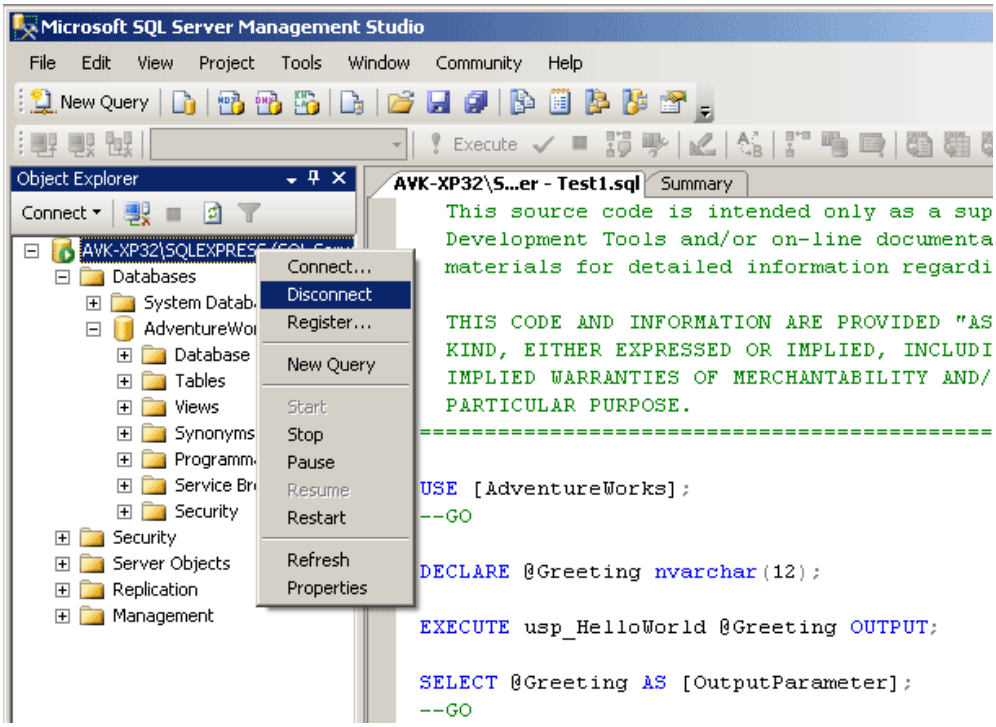
  - Press **OK** to save the changes.

3. The Profiler Run

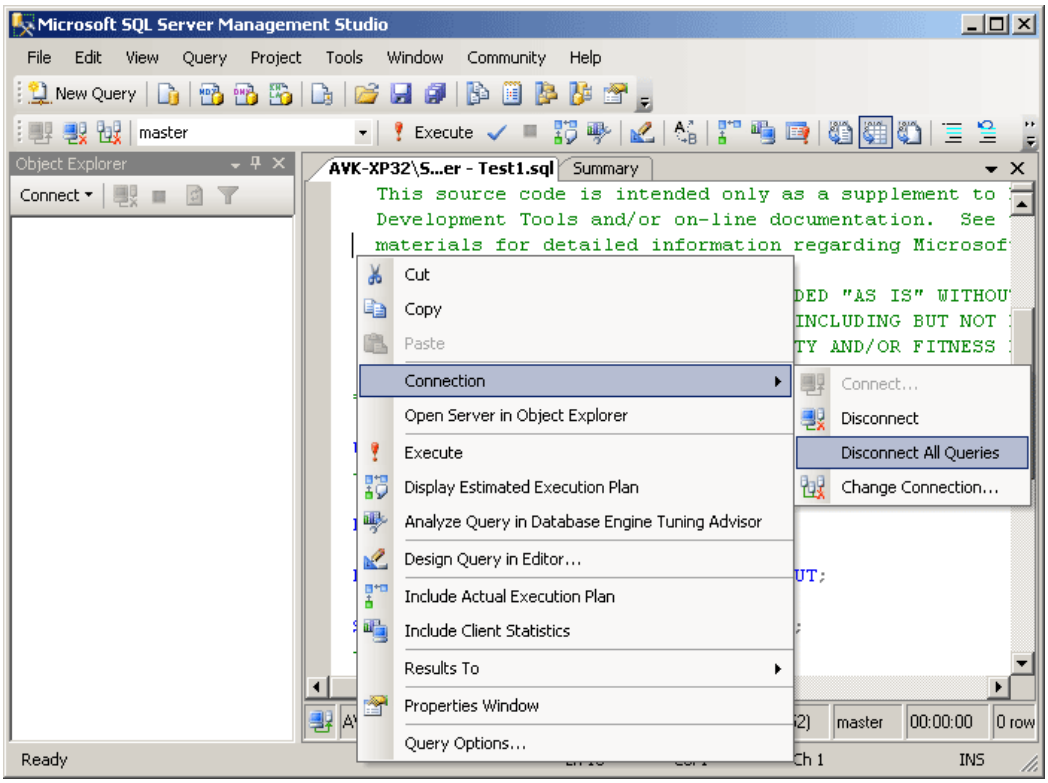
Prepare for the run:

- Disconnect SQL Server Management Studio and all queries from your SQL Server instance:
  - Open the **Object Explorer** panel of SQL Server Management Studio.

- Right-click the server node and choose **Disconnect** from the context menu.

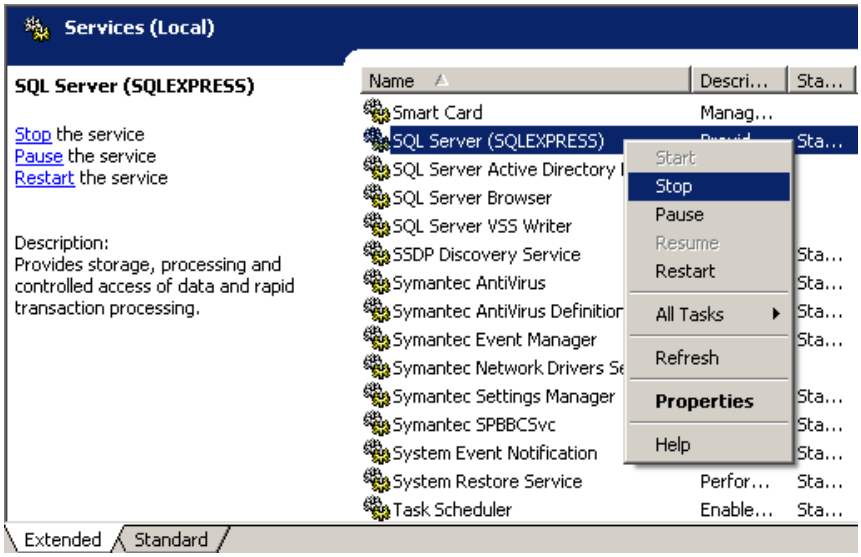


- If you have queries that are open in the Management Studio and that are connected to the server, then you should disconnect these queries. To do this, right-click somewhere within a query editor and choose **Connections | Disconnect All Queries** from the context menu.



- Stop the SQL Server service:

- Open the **Control Panel | Administrative Tools | Services** window.
- Right-click the **SQL Server (SQLSERVER2005)** item (**SQL Server (SQLEXPRESS)** if your are running SQL Server Express Edition) and choose **Stop** from the context menu.



- Switch to AQtime. Check that Normal mode is selected and that the project is prepared as it is described above.

Now we can run the profiler:

- Start profiling as you normally would.
- Switch to SQL Server Management Studio and open the SQL code, which calls the routines from your CLR integration assembly, in it.
- Right-click the SQL code in the editor and choose **Connection | Change Connection** from the context menu.

  In the ensuing dialog, connect to your SQL Server instance.
- Right-click within the editor and choose **Execute** from the context menu. This will execute the SQL code.
- To generate profiling results, choose the **Run | Get Results** item from AQtime's main menu or press **Get Results** on the Standard toolbar. (If you use AQtime integrated into Microsoft Visual Studio, then to generate profiling results, choose the **Profile | Get Results** item from Visual Studio's main menu or press **Get Results** on the **AQtime** toolbar. If you use Borland Developer Studio, then to generate profiling results, choose **Get Results** on Borland Developer Studio's **Run.AQtime** toolbar.)

  AQtime will display the results in its panels.

To terminate the profiler run:

- Disconnect all queries from the SQL Server.
- To stop the profiler, click **Terminate** on the Standard toolbar (If you use AQtime intergrated into Micrsofot Visual Studio, select **Profile | Terminate** from Visual Studio's main menu. In Borland Developer Studio, click **Terminate** on the **Run.AQtime** toolbar.)

Since we selected the *Profile Entire .NET Code* box, the profiling results contain all managed routines that were executed during the profiler run. To find the routines that belong to your assembly, you can group results by class name. You can also sort results to find the desired routines faster.

You will find that the Module Name column does not display any values for your routines. This happens because the module name cannot be determined.

Do not forget to start the SQL Server service after you finish the profiling sessions:

- You can do this in the Control Panel | Administrative Tools | Services window.
- Right-click the service in the window and choose **Start** from the context menu.

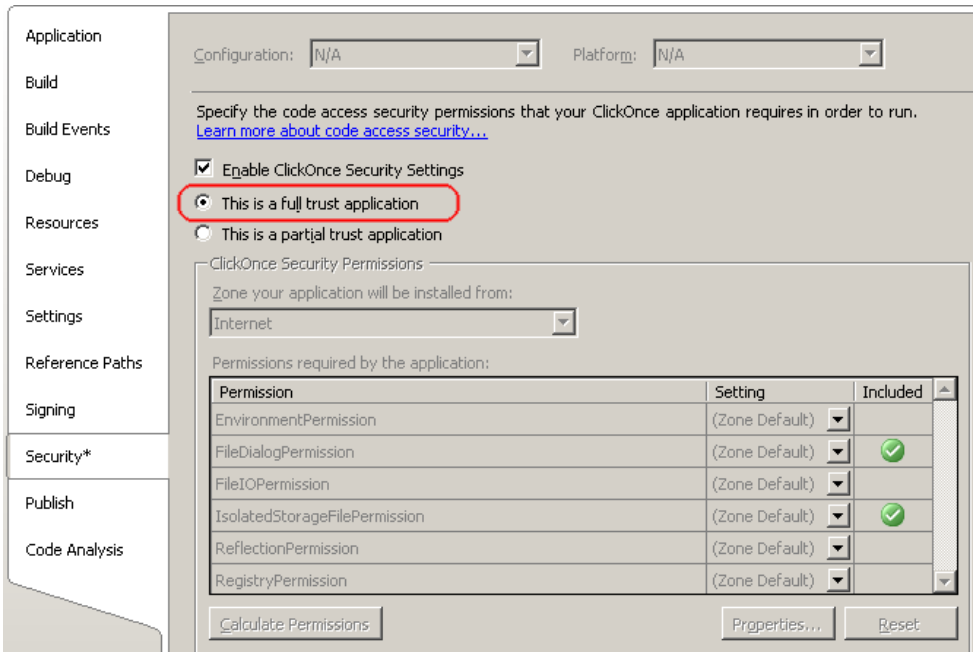
Profiling WPF Browser (XBAP) Applications

WPF Browser applications (or **XAML Browser** applications, or **XBAP**) are a specific kind of application that are compiled into .xbap extensions and can be run in Internet Explorer. This topic explains how you can profile your WPF Browser applications with AQtime.

1. Preparing the WPF Browser Application

In order for AQtime to be able to profile your WPF Browser application, you should specify the compiler settings:

1. Open your WPF Browser application in Visual Studio.
2. Right-click your project in the **Solution Explorer** and choose **Properties** from the context menu. This will open the project properties for editing.
3. Modify project settings to compile the application with debug information (see *Compiler Settings for Microsoft Visual C# 2005 and 2008* and *Compiler Settings for Microsoft Visual Basic 2005 and 2008*).
4. In the project properties editor, activate the **Security** tabbed page and select **This is a full trust application**:

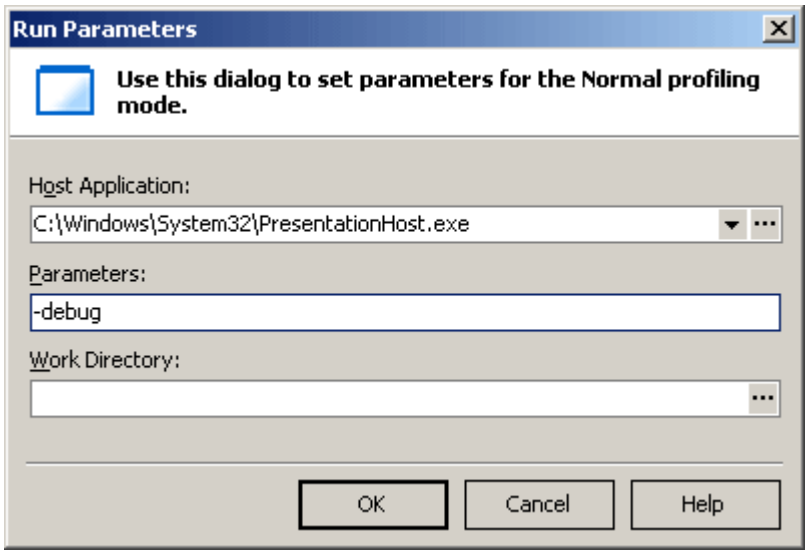


5. Save the settings and re-build the application.

2. Preparing the AQtime Project

1. Launch AQtime and create a new empty AQtime project.
2. Select the **Normal** profiling mode on AQtime's **Standard** toolbar (from the **AQtime** toolbar, if you use AQtime integrated into Visual Studio, or from the **Run.AQtime** toolbar in BDS).
3. Choose **Run | Parameters** from AQtime's main menu (or **Profile | Parameters** from Visual Studio or BDS menu, if you use the intergrated version of AQtime). This will invoke the Run Parameters dialog.
4. Specify the following values in the dialog's edit boxes:

   Host Application: `C:\Windows\System32\PresentationHost.exe`
   Parameters: `-debug`



Click **OK** to save the changes and to close the dialog.

5. Switch to AQtime's **Setup** panel and add your application's executable (.exe) to the AQtime project.

   Typically, the .exe module resides in the *<Application Folder>\Bin\Debug* folder. However, if you published and launched your application, then it will be placed in the cache.

   - On Windows XP computers the cache is located here:

     <Documents and Settings>\<USER_NAME>\Local Settings\Apps\2.0

   - On Windows Vista, Windows Server 2008 or Windows 7 machines, the cache is located here:

     <Users>\<USER_NAME>\AppData\Local\Apps\2.0

6. Now the project is ready for profiling. If needed, you can now create profiling areas and tune triggers and actions.

3. Profiling

1. Start profiling in AQtime. AQtime will launch PresentationHost, which, in turn, starts your application.

2. Open Windows Explorer, go to the folder that contains the compiled version of your WPF Browser application and double-click the application (the .xbap file). This will launch the application.
3. Perform the desired actions over the application. AQtime will profile it.
4. To generate results, either close the application, or use the Get Results command (see *Getting Results During Testing*).

   AQtime will display profiling results in its panels.

Tip: There is another variant to launch an XBAP application from AQtime: when you modify the launch settings in the Run Parameters dialog, enter the following value into the **Parameters** box:

```
-debug <Your_XBAP_Module_Name_and_Path>
```

In this case, the .xbap application will be launched automatically when you start profiling.

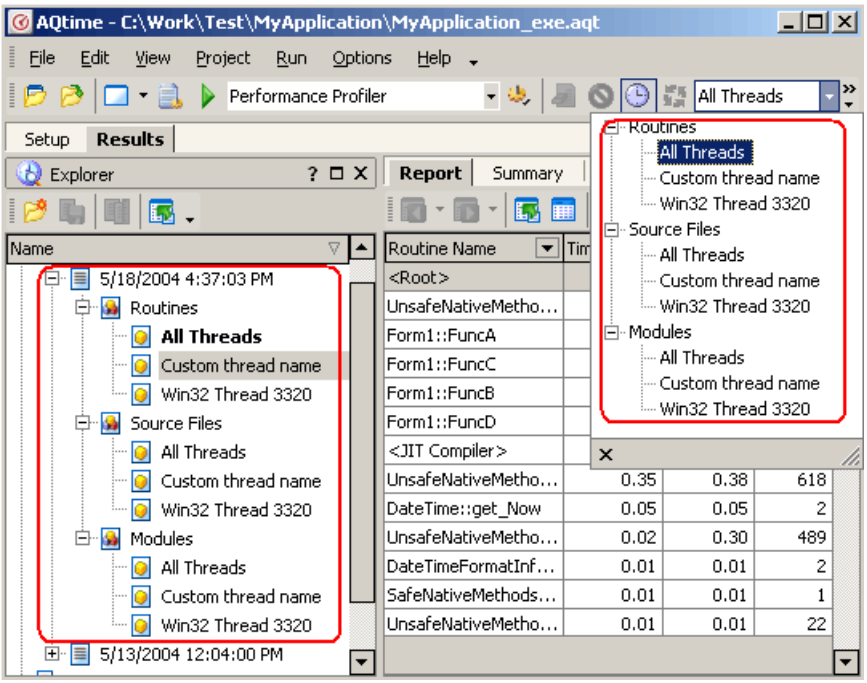
If the name or path of your .xbap module contains spaces, enclose the parameter in quotes, for instance:

```
-debug "C:\My Downloads\MyXBAPApp.xbap"
```

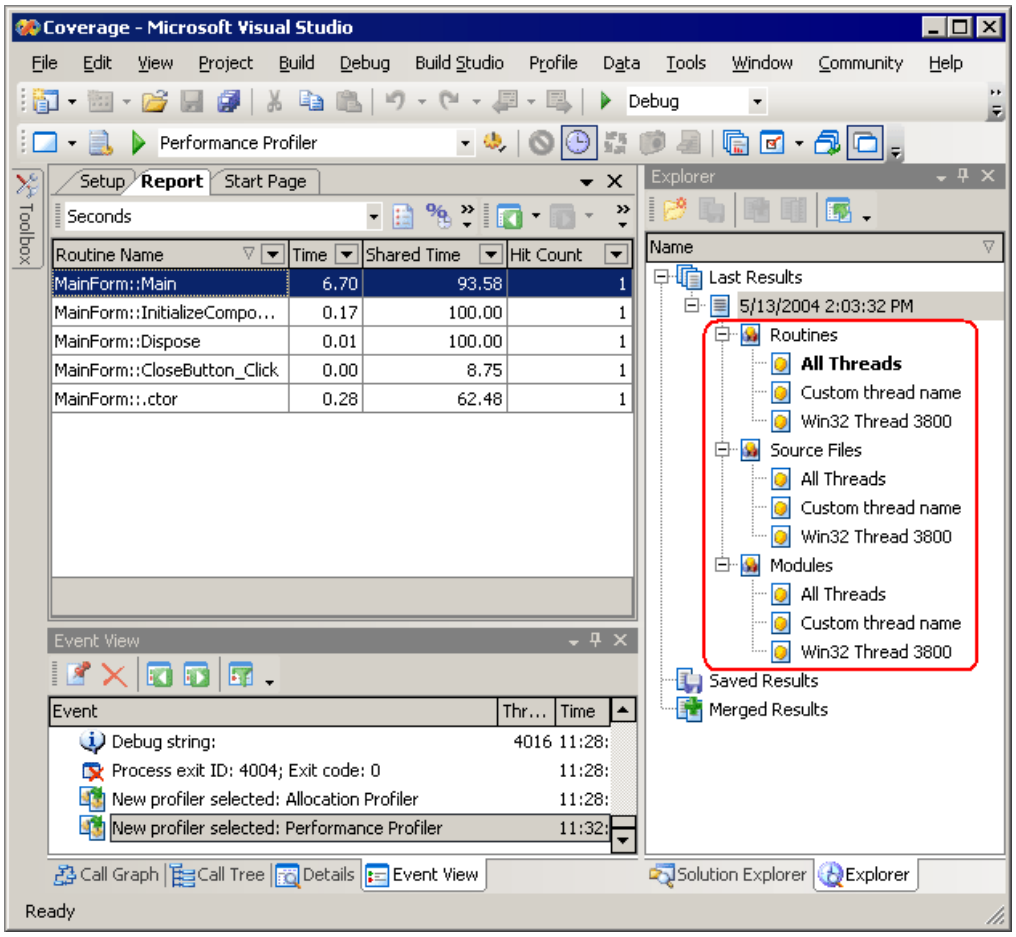
Profiling Multiple Threads

AQtime profilers trace function calls in all application threads. This is quite transparent for the user and does not require special preparation for the application. However, to obtain correct profiling results and the stack of functions, you may need to assign certain values to the *Thread model* option of the selected profiler. This topic describes the thread profiling functionality of AQtime and describes why the option is needed.

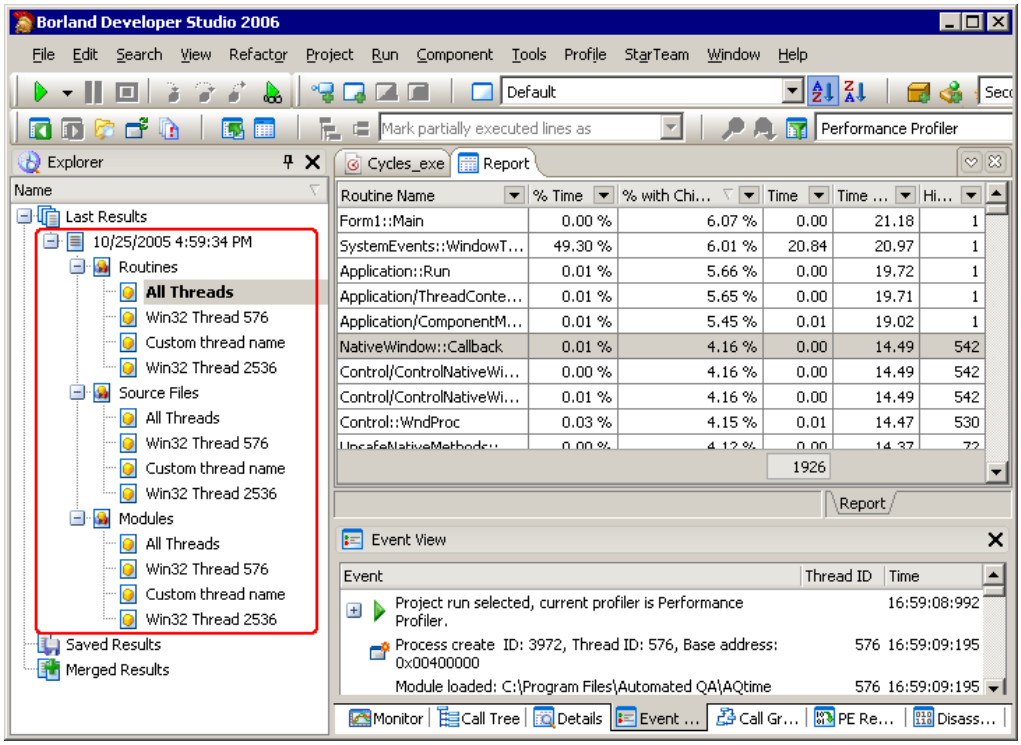
AQtime's Allocation, BDE SQL, Coverage, Function Trace, Performance, Reference Count and Resource profilers log and save results by threads. If the profile run includes more than one thread, you can select a single thread to show in the Report panel from the Explorer panel or from the **Result Items** toolbar item (by default, this item is hidden). Of course, you can display all of threads together as well:



AQtime standalone



AQtime integrated into Visual Studio


Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Default
Mark partially executed lines as
Performance Profiler
Explorer
Name
Last Results
10/25/2005 4:59:34 PM
Routines
All Threads
Win32 Thread 576
Custom thread name
Win32 Thread 2536
Source Files
All Threads
Win32 Thread 576
Custom thread name
Win32 Thread 2536
Modules
All Threads
Win32 Thread 576
Custom thread name
Win32 Thread 2536
Saved Results
Merged Results
Cycles_exe
Report
Routine Name % Time % with Chi... Time Time ... Hi...
Form1::Main 0.00 % 6.07 % 0.00 21.18 1
SystemEvents::WindowT... 49.30 % 6.01 % 20.84 20.97 1
Application::Run 0.01 % 5.66 % 0.00 19.72 1
Application/ThreadConte... 0.01 % 5.65 % 0.00 19.71 1
Application/ComponentM... 0.01 % 5.45 % 0.01 19.02 1
NativeWindow::Callback 0.01 % 4.16 % 0.00 14.49 542
Control/ControlNativeWi... 0.00 % 4.16 % 0.00 14.49 542
Control/ControlNativeWi... 0.01 % 4.16 % 0.00 14.49 542
Control::WndProc 0.03 % 4.15 % 0.01 14.47 530
1926
Report
Event View
Event Thread ID Time
Project run selected, current profiler is Performance Profiler. 16:59:08:992
Process create ID: 3972, Thread ID: 576, Base address: 0x00400000 576 16:59:09:195
Module loaded: C:\Program Files\Automated QA\AQtime 576 16:59:09:195
Monitor Call Tree Details Event ... Call Gr... PE Re... Disass...

AQtime integrated into Borland Developer Studio

As you can see, threads are identified by their IDs. For .NET applications running under the .NET Framework version 2.0 or later, AQtime is able to obtain user-defined names of CLR threads that are assigned through the `Name` property of the `Thread` class (for more information, see the property description in the MSDN Library).

AQtime also lets you assign descriptive names to threads from your application. These names will be used instead of the default thread names in AQtime panels and toolbar items. As you can see on the picture above, one of the threads has a user-defined name. For more information, see *Assigning Names to Threads*.

The Performance and Coverage profilers can also be controlled by triggers, which turn profiling on or off for a particular thread in which a function (the trigger) runs. This is an essential tool to winnow out profile information from complex multithreaded applications.

Note that a thread is not a process. If a profiled application launches a new process, it will simply escape being profiled by AQtime, which can only watch over the child process it has launched itself (or to which it attached).

We would like to point out that the operating system threads differ from the managed threads that may exist in .NET applications. Managed threads are controlled by the common language runtime. It can create one or several operating system threads to run a single managed thread. AQtime profilers include the *Thread model* option that specifies what thread model the profiler should log the results. If you select the ***Win32 Threads*** value for this option, the Performance and Coverage profilers will group results by the operating system threads. The Function Trace, Allocation and Resource profilers will trace the stack of function calls by operating system threads. If you select ***CLR Threads***, the profilers will gather results and trace the call stack by managed threads. The profiling results for these threads are displayed in the same way as results of the operating system threads: the Explorer panel holds a list of the threads (but these are not operating system threads, these are managed threads). You can view results for a thread by selecting this thread in the Explorer panel or in the Threads dropdown list.

There is one more value for the Thread model option: *COM Threads*. It means that AQtime should analyze logical threads that occur when a COM server works with several COM clients simultaneously. To keep this topic in bounds, we described these threads in a separate topic, *Profiling COM Logical Threads*.

Profiling Multiple Processes

One instance of AQtime can only profile one process. So, to profile several processes, you have to run several AQtime instances - one instance per process.

In general, profiling of multiple processes does not differ from profiling a single process. For each process you create a project, add the desired module(s) to it, specify the profiling areas, triggers, actions and other settings and then either start the process from AQtime or attach to the process using the "Attach to Process" feature. For instance, you can use the "Attach to Process" functionality, if one of the processes is started by another process. Note, however, that attaching requires time, so you will not be able to profile the startup code of the started process.

To solve this problem, you can modify the registry settings so that when the operating system gets a command to start a process, it automatically launches an AQtime instance, which you can then use to profile your application. To modify the registry settings:

- Launch the Registry Editor (regedit).
- Open the **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options** node.
- Add a key for your application to this node:
  - Right-click the node and choose **New | Key** from the context menu. This will create a new key node under the Image File Execution Options node.
  - Right-click the new node and select **Rename** from the context menu.

    Change the key name to the name of your executable (for instance, *MyApplication.exe*). Only specify the file name and extension. The path is not needed.
- Specify AQtime as a debugger for the application. To do this:
  - Right-click the new key node and select **New | String Value** from the context menu. This will append a new string value to the created key.
  - Switch to the right panel of the Registry Editor, right-click the string value's node and choose **Rename** from the context menu.

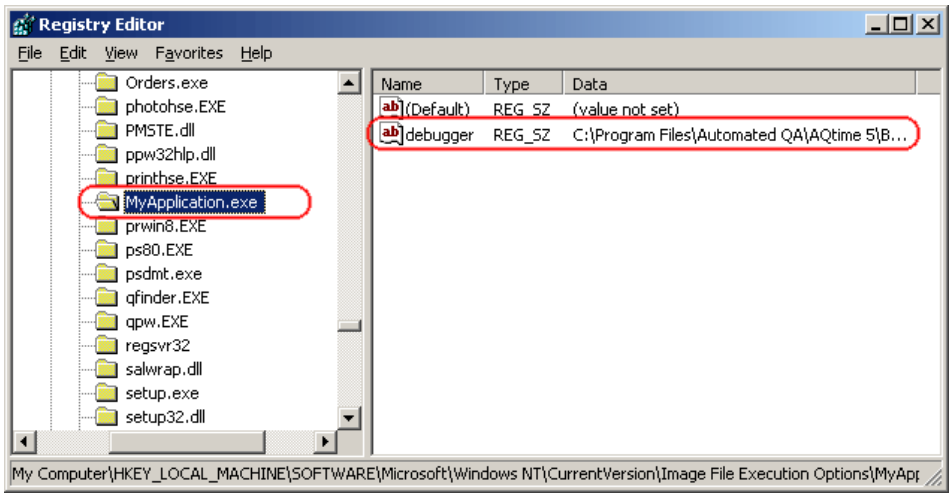
    In the ensuing in-place editor, change the value name to **debugger**.
  - Right-click the value node again and select **Modify** from the context menu. This will call the **Edit String** dialog.
  - In the **Value** data box of the dialog, enter the fully qualified name of AQtime.exe, for instance, *C:\Program Files\Automated QA\AQtime 5\Bin\AQtime.exe*.
  - Click **OK** to save the changes and to close the dialog.

Now the operating system will launch AQtime every time your application starts.

Do not forget to remove the registry key, when the automatic start of AQtime is no longer needed.

If you need to enable or disable the automatic launch frequently, you can create two .reg files:

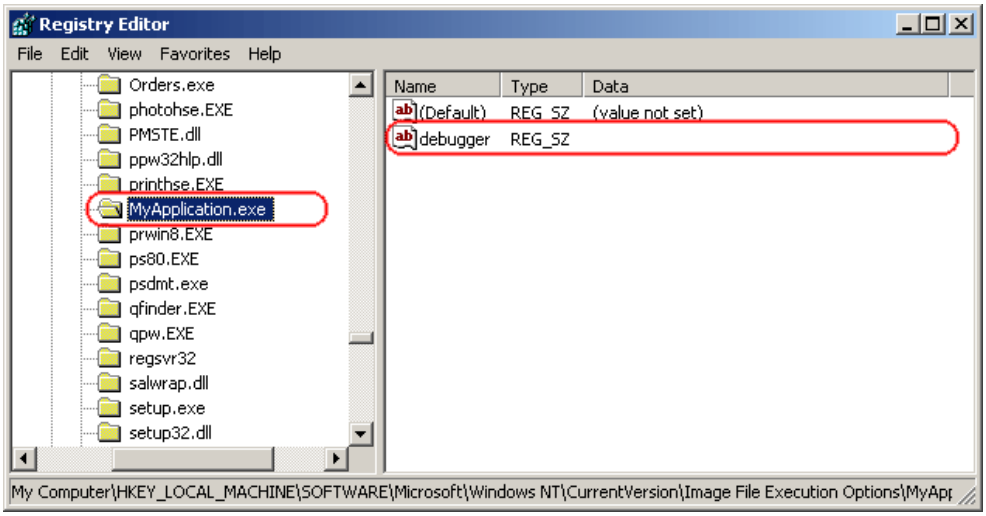
- One of the files will include the Registry key and the debugger value with the path to AQtime:



To create this file:

9. Modify the registry as it was described above.
10. Select the key in the tree view on the left of the Registry Editor.
11. Choose **File | Export** from the editor's main menu and specify the file name in the ensuing **Export Registry File** dialog.

- Another file will store the Registry key and the debugger value, but it will not store the path to AQtime:



To create this file:

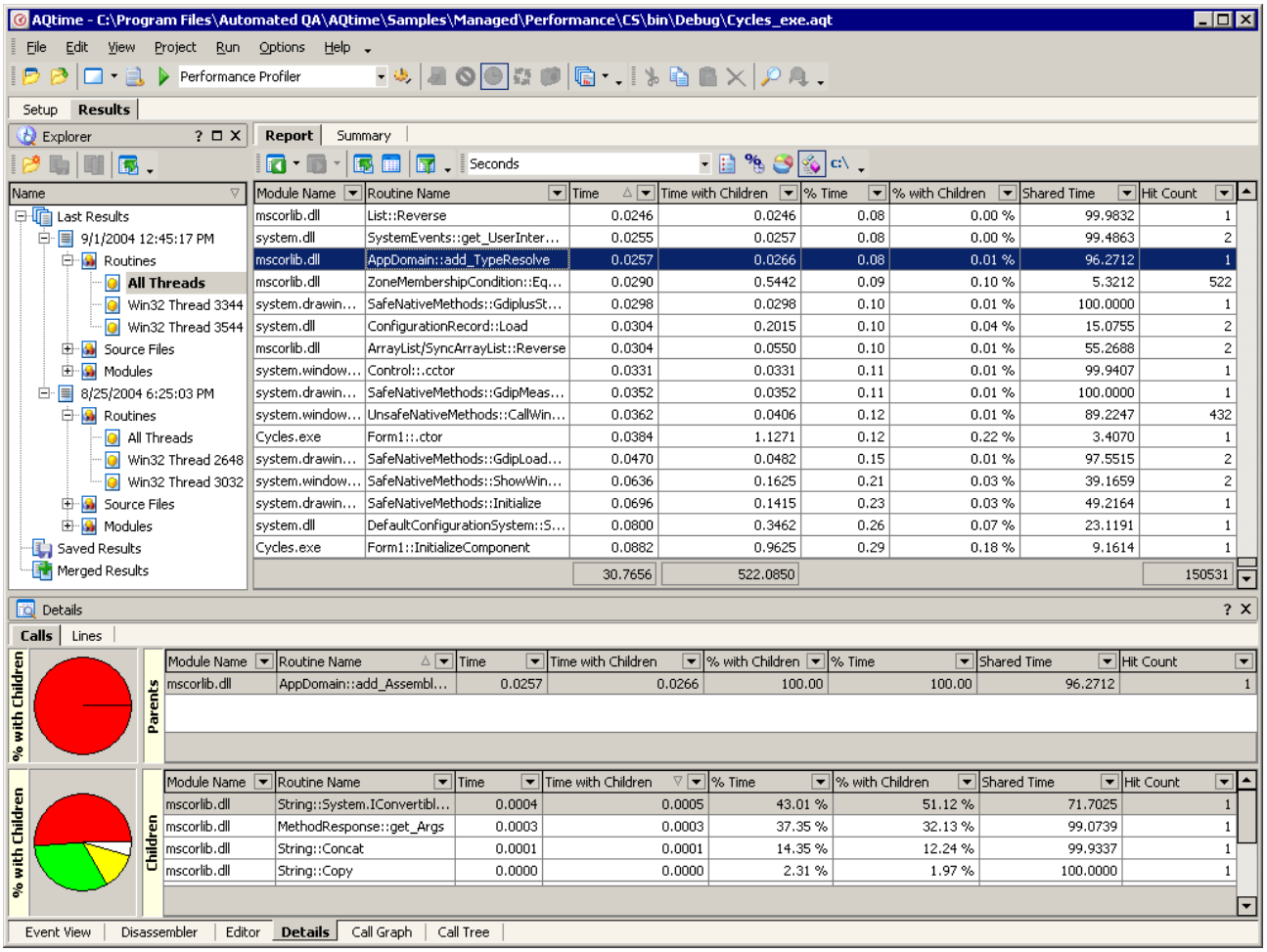
12. Modify the registry as it was described above.
13. Clear the data of the **debugger** value.
14. Export the key to a file by choosing **File | Export** from the main menu of the Registry Editor.

Now, when you need to enable or disable the automatic launch of AQtime for your application, you can execute one of the .reg files.

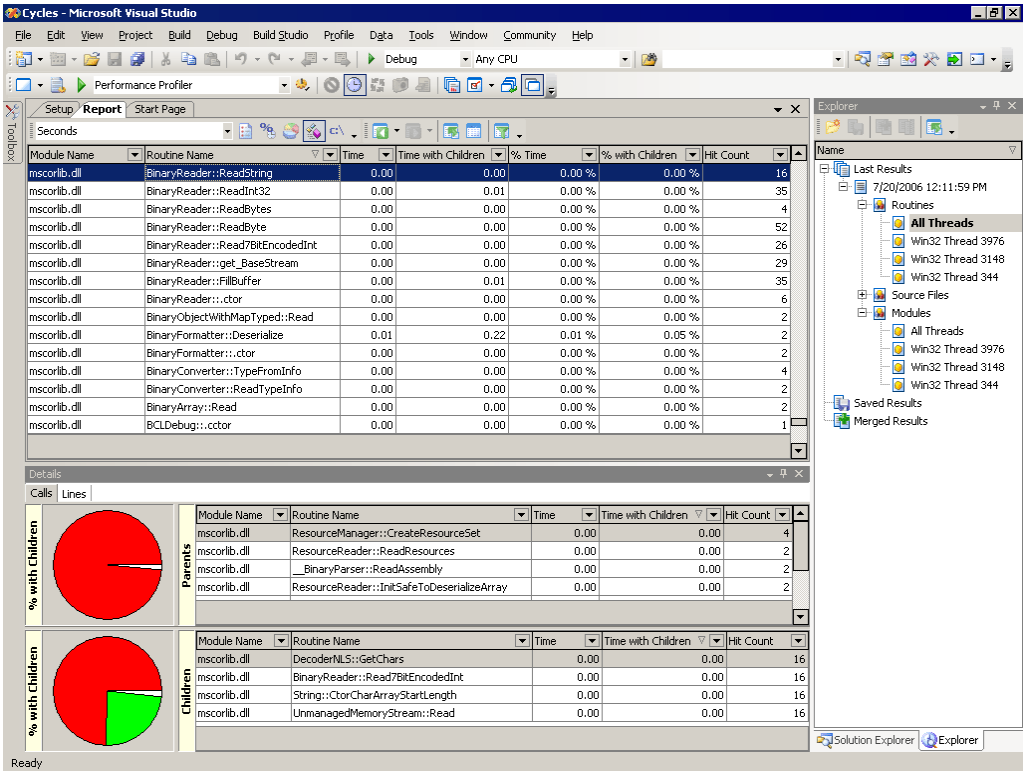
Profiling System Calls

Your applications can use system libraries specific for your operating system (native API) or, if your application is a .NET application, it can use system libraries (assemblies) shared across the .NET Framework platform. Generally, when profiling applications in AQtime, system calls are not traced. This is simply because the system libraries have not been added to the Setup panel.

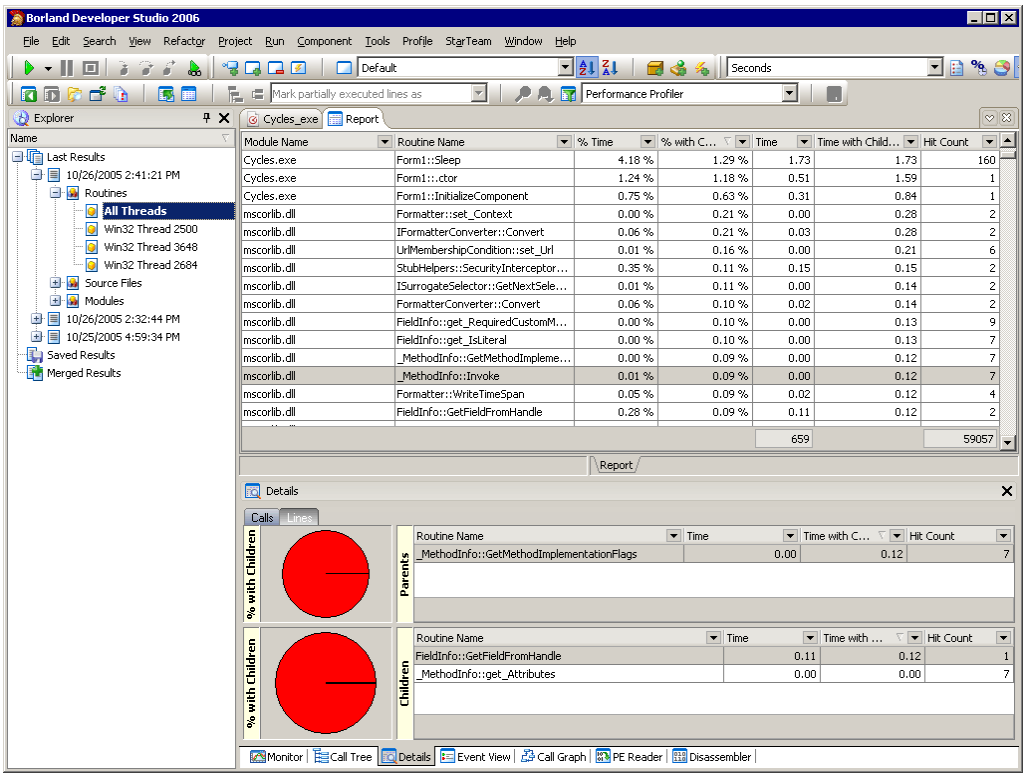
To profile routines from a system dynamic link library or assembly, simply add this library (assembly) to your AQtime project by using **Add Module** (**Add Assembly**) in the context menu or the toolbar of the Setup panel. Run the desired profiler then and you will see functions from these libraries in the profiling results. Note that AQtime profilers will report on the system functions just as they report on application functions. You will be able to examine them with the full range of AQtime panels:


AQtime - C:\Program Files\Automated QA\AQtime\Samples\Managed\Performance\CS\bin\Debug\Cycles_exe.aqt
File Edit View Project Run Options Help
Performance Profiler
Setup Results
Explorer
Report Summary
Seconds
Name
Last Results
9/1/2004 12:45:17 PM
Routines
All Threads
Win32 Thread 3344
Win32 Thread 3544
Source Files
Modules
8/25/2004 6:25:03 PM
Routines
All Threads
Win32 Thread 2648
Win32 Thread 3032
Source Files
Modules
Saved Results
Merged Results
Module Name | Routine Name | Time | Time with Children | % Time | % with Children | Shared Time | Hit Count
mscorlib.dll | List::Reverse | 0.0246 | 0.0246 | 0.08 | 0.00 % | 99.9832 | 1
system.dll | SystemEvents::get_UserInter... | 0.0255 | 0.0257 | 0.08 | 0.00 % | 99.4863 | 2
mscorlib.dll | AppDomain::add_TypeResolve | 0.0257 | 0.0266 | 0.08 | 0.01 % | 96.2712 | 1
mscorlib.dll | ZoneMembershipCondition::Eq... | 0.0290 | 0.5442 | 0.09 | 0.10 % | 5.3212 | 522
system.drawin... | SafeNativeMethods::GdiplusSt... | 0.0298 | 0.0298 | 0.10 | 0.01 % | 100.0000 | 1
system.dll | ConfigurationRecord::Load | 0.0304 | 0.2015 | 0.10 | 0.04 % | 15.0755 | 2
mscorlib.dll | ArrayList/SyncArrayList::Reverse | 0.0304 | 0.0550 | 0.10 | 0.01 % | 55.2688 | 2
system.window... | Control::.cctor | 0.0331 | 0.0331 | 0.11 | 0.01 % | 99.9407 | 1
system.drawin... | SafeNativeMethods::GdipMeas... | 0.0352 | 0.0352 | 0.11 | 0.01 % | 100.0000 | 1
system.window... | UnsafeNativeMethods::CallWin... | 0.0362 | 0.0406 | 0.12 | 0.01 % | 89.2247 | 432
Cycles.exe | Form1::.ctor | 0.0384 | 1.1271 | 0.12 | 0.22 % | 3.4070 | 1
system.drawin... | SafeNativeMethods::GdipLoad... | 0.0470 | 0.0482 | 0.15 | 0.01 % | 97.5515 | 2
system.window... | SafeNativeMethods::ShowWin... | 0.0636 | 0.1625 | 0.21 | 0.03 % | 39.1659 | 2
system.drawin... | SafeNativeMethods::Initialize | 0.0696 | 0.1415 | 0.23 | 0.03 % | 49.2164 | 1
system.dll | DefaultConfigurationSystem::S... | 0.0800 | 0.3462 | 0.26 | 0.07 % | 23.1191 | 1
Cycles.exe | Form1::InitializeComponent | 0.0882 | 0.9625 | 0.29 | 0.18 % | 9.1614 | 1
30.7656 522.0850 150531
Details
Calls Lines
% with Children
Parents
Module Name | Routine Name | Time | Time with Children | % with Children | % Time | Shared Time | Hit Count
mscorlib.dll | AppDomain::add_Assembl... | 0.0257 | 0.0266 | 100.00 | 100.00 | 96.2712 | 1
% with Children
Children
Module Name | Routine Name | Time | Time with Children | % Time | % with Children | Shared Time | Hit Count
mscorlib.dll | String::System.IConvertibl... | 0.0004 | 0.0005 | 43.01 % | 51.12 % | 71.7025 | 1
mscorlib.dll | MethodResponse::get_Args | 0.0003 | 0.0003 | 37.35 % | 32.13 % | 99.0739 | 1
mscorlib.dll | String::Concat | 0.0001 | 0.0001 | 14.35 % | 12.24 % | 99.9337 | 1
mscorlib.dll | String::Copy | 0.0000 | 0.0000 | 2.31 % | 1.97 % | 100.0000 | 1
Event View Disassembler Editor Details Call Graph Call Tree

AQtime standalone


Cycles - Microsoft Visual Studio
File Edit View Project Build Debug Build Studio Profile Data Tools Window Community Help
Debug
Any CPU
Performance Profiler
Setup Report Start Page
Seconds
Module Name Routine Name Time Time with Children % Time % with Children Hit Count
mscorlib.dll BinaryReader::ReadString 0.00 0.00 0.00 % 0.00 % 16
mscorlib.dll BinaryReader::ReadInt32 0.00 0.01 0.00 % 0.00 % 35
mscorlib.dll BinaryReader::ReadBytes 0.00 0.00 0.00 % 0.00 % 4
mscorlib.dll BinaryReader::ReadByte 0.00 0.00 0.00 % 0.00 % 52
mscorlib.dll BinaryReader::Read7BitEncodedInt 0.00 0.00 0.00 % 0.00 % 26
mscorlib.dll BinaryReader::get_BaseStream 0.00 0.00 0.00 % 0.00 % 29
mscorlib.dll BinaryReader::FillBuffer 0.00 0.01 0.00 % 0.00 % 35
mscorlib.dll BinaryReader::.ctor 0.00 0.00 0.00 % 0.00 % 6
mscorlib.dll BinaryObjectWithMapTyped::Read 0.00 0.00 0.00 % 0.00 % 2
mscorlib.dll BinaryFormatter::Deserialize 0.01 0.22 0.01 % 0.05 % 2
mscorlib.dll BinaryFormatter::.ctor 0.00 0.00 0.00 % 0.00 % 2
mscorlib.dll BinaryConverter::TypeFromInfo 0.00 0.00 0.00 % 0.00 % 4
mscorlib.dll BinaryConverter::ReadTypeInfo 0.00 0.00 0.00 % 0.00 % 2
mscorlib.dll BinaryArray::Read 0.00 0.00 0.00 % 0.00 % 2
mscorlib.dll BCLDebug::.cctor 0.00 0.00 0.00 % 0.00 % 1
Details
Calls Lines
% with Children
Parents
Module Name Routine Name Time Time with Children Hit Count
mscorlib.dll ResourceManager::CreateResourceSet 0.00 0.00 4
mscorlib.dll ResourceReader::ReadResources 0.00 0.00 2
mscorlib.dll __BinaryParser::ReadAssembly 0.00 0.00 2
mscorlib.dll ResourceReader::InitSafeToDeserializeArray 0.00 0.00 2
% with Children
Children
Module Name Routine Name Time Time with Children Hit Count
mscorlib.dll DecoderNLS::GetChars 0.00 0.00 16
mscorlib.dll BinaryReader::Read7BitEncodedInt 0.00 0.00 16
mscorlib.dll String::CtorCharArrayStartLength 0.00 0.00 16
mscorlib.dll UnmanagedMemoryStream::Read 0.00 0.00 16
Explorer
Name
Last Results
7/20/2006 12:11:59 PM
Routines
All Threads
Win32 Thread 3976
Win32 Thread 3148
Win32 Thread 344
Source Files
Modules
All Threads
Win32 Thread 3976
Win32 Thread 3148
Win32 Thread 344
Saved Results
Merged Results
Solution Explorer
Explorer
Ready

AQtime integrated into Visual Studio


Borland Developer Studio 2006
File Edit Search View Refactor Project Run Component Tools Profile StarTeam Window Help
Default
Seconds
Mark partially executed lines as
Performance Profiler
Explorer
Name
Last Results
10/26/2005 2:41:21 PM
Routines
All Threads
Win32 Thread 2500
Win32 Thread 3648
Win32 Thread 2684
Source Files
Modules
10/26/2005 2:32:44 PM
10/25/2005 4:59:34 PM
Saved Results
Merged Results
Cycles_exe
Report
Module Name Routine Name % Time % with C... Time Time with Child... Hit Count
Cycles.exe Form1::Sleep 4.18 % 1.29 % 1.73 1.73 160
Cycles.exe Form1::.ctor 1.24 % 1.18 % 0.51 1.59 1
Cycles.exe Form1::InitializeComponent 0.75 % 0.63 % 0.31 0.84 1
mscorlib.dll Formatter::set_Context 0.00 % 0.21 % 0.00 0.28 2
mscorlib.dll IFormatterConverter::Convert 0.06 % 0.21 % 0.03 0.28 2
mscorlib.dll UrlMembershipCondition::set_Url 0.01 % 0.16 % 0.00 0.21 6
mscorlib.dll StubHelpers::SecurityInterceptor... 0.35 % 0.11 % 0.15 0.15 2
mscorlib.dll ISurrogateSelector::GetNextSele... 0.01 % 0.11 % 0.00 0.14 2
mscorlib.dll FormatterConverter::Convert 0.06 % 0.10 % 0.02 0.14 2
mscorlib.dll FieldInfo::get_RequiredCustomM... 0.00 % 0.10 % 0.00 0.13 9
mscorlib.dll FieldInfo::get_IsLiteral 0.00 % 0.10 % 0.00 0.13 7
mscorlib.dll _MethodInfo::GetMethodImpleme... 0.00 % 0.09 % 0.00 0.12 7
mscorlib.dll _MethodInfo::Invoke 0.01 % 0.09 % 0.00 0.12 7
mscorlib.dll Formatter::WriteTimeSpan 0.05 % 0.09 % 0.02 0.12 4
mscorlib.dll FieldInfo::GetFieldFromHandle 0.28 % 0.09 % 0.11 0.12 2
659 59057
Report
Details
Calls Lines
% with Children
Parents
Routine Name Time Time with C... Hit Count
_MethodInfo::GetMethodImplementationFlags 0.00 0.12 7
% with Children
Children
Routine Name Time Time with ... Hit Count
FieldInfo::GetFieldFromHandle 0.11 0.12 1
_MethodInfo::get_Attributes 0.00 0.00 7
Monitor Call Tree Details Event View Call Graph PE Reader Disassembler

AQtime integrated into Borland Developer Studio

In fact, the described method lets you profile routines from any dynamic link libraries, not just from system ones. The dynamic link libraries may be compiled without debug information, because when profiling calls to library routines, AQtime will use information from the tables of imported and exported functions that are included in each Windows executable. If the dynamic link libraries have no debug info, you can profile them at routine level only.

If your application is a .NET application, you can profile functions contained in .NET Framework assemblies in another manner: you can check the *Profile Entire .NET Code* box in the Setup panel. If this box is checked, AQtime will profile all assemblies whose routines are called during the profiler run. It will profile even those assemblies and routines that are not added to the profiling areas. This is especially useful, if you do not know exactly what libraries your application depends on. However, using AQtime you can easily determine which libraries it uses and then include them in the profiling tasks. See *Knowing What Libraries Your Application Uses*.

We would like to note one more time that the *Profile Entire .NET Code* box lets you profile only .NET Framework assemblies. If you would like to profile routines from the operating system DLLs, you should add these DLLs to profiling areas.

The .NET Framework assemblies call functions from the operating system libraries using special stub functions that prepare parameters and call functions from the operating system libraries. If you check *Profile Entire .NET Code* or if you add the .NET Framework assemblies to the Setup panel, AQtime will profile these stub functions. To profile functions from the operating system DLLs, add these DLLs to the Setup panel.

Profiling Managed and Unmanaged Code

Quite often .NET applications include **mixed code**, that is, certain parts of the application executable are in MSIL code (managed code) and other parts are native code (unmanaged code). AQtime can analyze both managed and unmanaged code. It does not require any special preparations for profiling of mixed code except cases when modules, which include unmanaged sections, must be compiled with debug information. For example, if the application itself includes both managed and unmanaged code, you have to compile it with debug information in order to profile its unmanaged sections. See *How AQtime Profilers Use Metadata and Debug Information*.

One of the typical examples of using mixed code is a .NET application that uses a COM object, which is implemented in a native-code module. In this instance, you have to compile the native-code module with debug information to profile its functions. The way in which you include debug information into the executable depends upon the compiler. See *How AQtime Profilers Use Metadata and Debug Information*.

We would like to note once again that AQtime can profile both managed and unmanaged code. For example, with AQtime you can profile both managed and unmanaged modules at line level, you can define triggers and actions using both managed and unmanaged functions, etc. The only exception from this rule is the *Profile Entire .NET Code* setting in the Setup panel. It works for managed code only.

AQtime includes sample applications that illustrate the profiling of mixed code:

.NET applications

Microsoft Visual Studio .NET projects

- <AQtime>\Samples\Managed\VS.NET\MixedProfiling\Managed\CS - Microsoft Visual C# .NET
- <AQtime>\Samples\Managed\VS.NET\MixedProfiling\Managed\VB - Microsoft Visual Basic .NET
- <AQtime>\Samples\Managed\VS.NET\MixedProfiling\Managed\VC - Microsoft Visual C++ .NET

- <AQtime>\Samples\Managed\VS.NET\MixedProfiling\Managed\JS - Microsoft Visual J# .NET

Microsoft Visual Studio 2005 projects

- <AQtime>\Samples\Managed\VS2005\MixedProfiling\Managed\CS - Microsoft Visual C# .NET
- <AQtime>\Samples\Managed\VS2005\MixedProfiling\Managed\VB - Microsoft Visual Basic .NET
- <AQtime>\Samples\Managed\VS2005\MixedProfiling\Managed\VC - Microsoft Visual C++ .NET
- <AQtime>\Samples\Managed\VS2005\MixedProfiling\Managed\JS - Microsoft Visual J# .NET

Native-code DLLs

- <AQtime>\Samples\Managed\VS.NET\MixedProfiling\Unmanaged\VC - Microsoft Visual C++ (Visual Studio 7.x project)
- <AQtime>\Samples\Managed\VS.NET\MixedProfiling\Unmanaged\Delphi - Borland Delphi
- <AQtime>\Samples\Managed\VS2005\MixedProfiling\Unmanaged\VC - Microsoft Visual C++ (Visual Studio 7.x project)
- <AQtime>\Samples\Managed\VS2005\MixedProfiling\Unmanaged\VC2005 - Microsoft Visual C++ (Visual Studio 2005 project)
- <AQtime>\Samples\Managed\VS2005\MixedProfiling\Unmanaged\Delphi - Borland Delphi

You can download compiled versions of these sample applications from http://www.automatedqa.com/downloads/aqtime/. For more information on how to profile these samples, see *Profiling Mixed Code Tutorial* in on-line help.

Profiling Recursive Routines

We will call any function that calls itself or that is eventually called by a child function *recursive*. In AQtime terms, a recursive function is one that belongs to its own descendants (children, grandchildren, etc.). For the Performance profiler, this poses an unavoidable problem regarding what AQtime should call *Time With Children* and what it should call *Time* (that is, without children) for this type of a function.

This topic explains the Time with Children problem for recursive functions, and the solution that is adopted by the Performance profiler.

Note: Time With Children present in the Performance profiler results only if you use the *Elapsed Time*, *User Time* or *User+Kernel Time* counters. If you profile your application with any other counter, for example, with *CPU Cache Misses* or *Hard Memory Page Faults*, Time With Children as well as other "Timing" results (such as Average Time, Min Time, etc.) will be replaced with similar values: Misses With Children, Faults with Children, Average Misses, Average Faults, etc. AQtime calculates these values in the same way it calculates the "Time" values. In further explanations we will use "Time", but since the results are similar, the explanations are also true for "Misses", "Faults" and other results.

Time and **Time with Children** apply not to one call, but to the sum of calls throughout the profile run. Now, imagine that one function, `FuncA`, calls itself three times in a row, so that the original call, `FuncA1`, gives rise to three more, `FuncA2`, `FuncA3` and `FuncA4`. Imagine also that `FuncA` takes 2 seconds to execute its own code. If these are the only calls during the profile run, Time should be 8 seconds. But Time with Children?

FuncA4 = 2

FuncA3 = 4

FuncA2 = 6

FuncA1 = 8

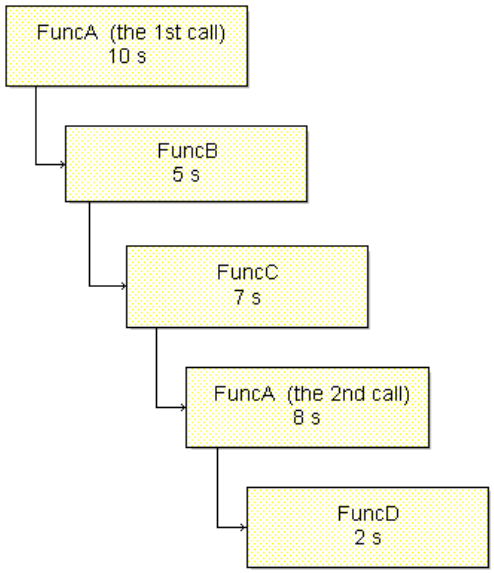
Total = 20 seconds

Now, imagine that the entire run was simply the original FuncA1 call. The entire run lasted 8 seconds, but Time with Children for FuncA is 20 seconds. This is grossly misleading. The reason is that one single execution, FuncA4, is counted separately as part of the child time for FunA3, FuncA2 and FuncA1, and it is also counted once as its "own" time - it is counted four times in all. Likewise, FuncA3 is counted three times and FuncA2 is counted twice. These repeat counts for the same actual execution bloat up Time with Children as soon as there is recursion.

In the very simple example above, we also know what solution we would like to see; Time with Children should be identical to the run time, 8 seconds. That is, the same as Time itself, since both values count exactly the same calls (FuncA1 through FuncA4).

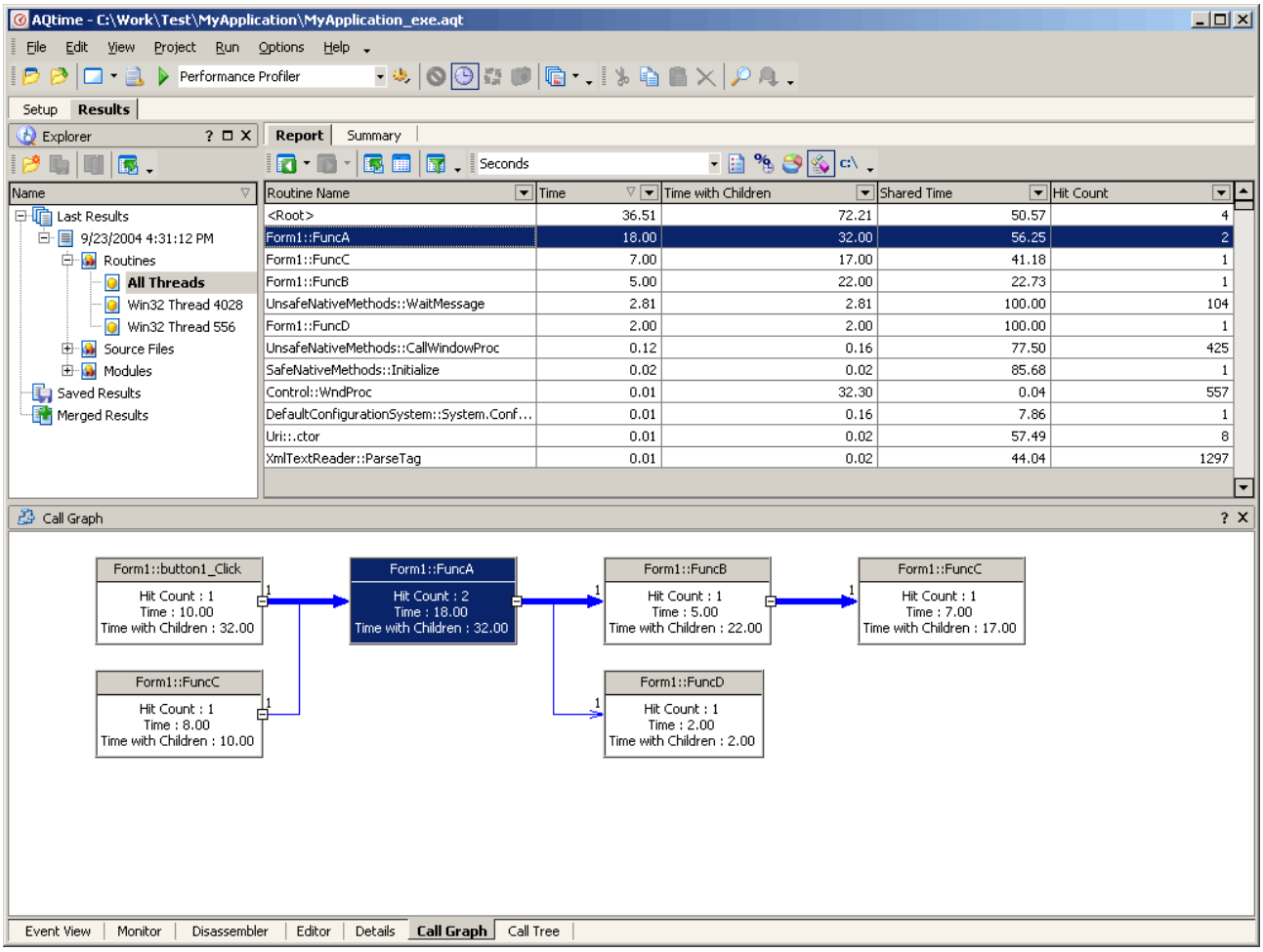
The Performance profiler detects the recursive calls and counts Time and Time with Children for recursive functions properly. With our simple FuncA example, this means FuncA2, FuncA3 and FuncA4 contribute nothing to Time with Children, so it remains what it was for FuncA1 alone, 8 seconds, the same as the run time. This is what we wanted.

Just to make sure everything is clear; here is a somewhat more complex example:



The timings are time spent executing the function itself.

The Performance profiler will report the following results:


AQtime - C:\Work\Test\MyApplication\MyApplication_exe.aqt
File Edit View Project Run Options Help
Performance Profiler
Setup Results
Explorer
Report Summary
Seconds
Name
Last Results
9/23/2004 4:31:12 PM
Routines
All Threads
Win32 Thread 4028
Win32 Thread 556
Source Files
Modules
Saved Results
Merged Results
Routine Name | Time | Time with Children | Shared Time | Hit Count
<Root> | 36.51 | 72.21 | 50.57 | 4
Form1::FuncA | 18.00 | 32.00 | 56.25 | 2
Form1::FuncC | 7.00 | 17.00 | 41.18 | 1
Form1::FuncB | 5.00 | 22.00 | 22.73 | 1
UnsafeNativeMethods::WaitMessage | 2.81 | 2.81 | 100.00 | 104
Form1::FuncD | 2.00 | 2.00 | 100.00 | 1
UnsafeNativeMethods::CallWindowProc | 0.12 | 0.16 | 77.50 | 425
SafeNativeMethods::Initialize | 0.02 | 0.02 | 85.68 | 1
Control::WndProc | 0.01 | 32.30 | 0.04 | 557
DefaultConfigurationSystem::System.Conf... | 0.01 | 0.16 | 7.86 | 1
Uri::.ctor | 0.01 | 0.02 | 57.49 | 8
XmlTextReader::ParseTag | 0.01 | 0.02 | 44.04 | 1297
Call Graph
Form1::button1_Click
Hit Count : 1
Time : 10.00
Time with Children : 32.00
Form1::FuncA
Hit Count : 2
Time : 18.00
Time with Children : 32.00
Form1::FuncB
Hit Count : 1
Time : 5.00
Time with Children : 22.00
Form1::FuncC
Hit Count : 1
Time : 7.00
Time with Children : 17.00
Form1::FuncC
Hit Count : 1
Time : 8.00
Time with Children : 10.00
Form1::FuncD
Hit Count : 1
Time : 2.00
Time with Children : 2.00
Event View Monitor Disassembler Editor Details Call Graph Call Tree

The net time of FuncA is about 18 seconds (10 seconds for the first call plus 8 seconds for the second call). FuncD and FuncB are child routines of FuncA. The Time and Time with Children results for them corresponds to the actual execution time of these routines. FuncC is a parent routine for FuncA. As we can see, the net time of the FuncA call, which was made within FuncC, is about 8 seconds and Time with Children is 10 seconds. These values also match to the actual time of function calls.

Note: If the first call to FuncA is not profiled for any reason (for instance, it is excluded by an off-trigger), the Performance profiler detects no recursion.

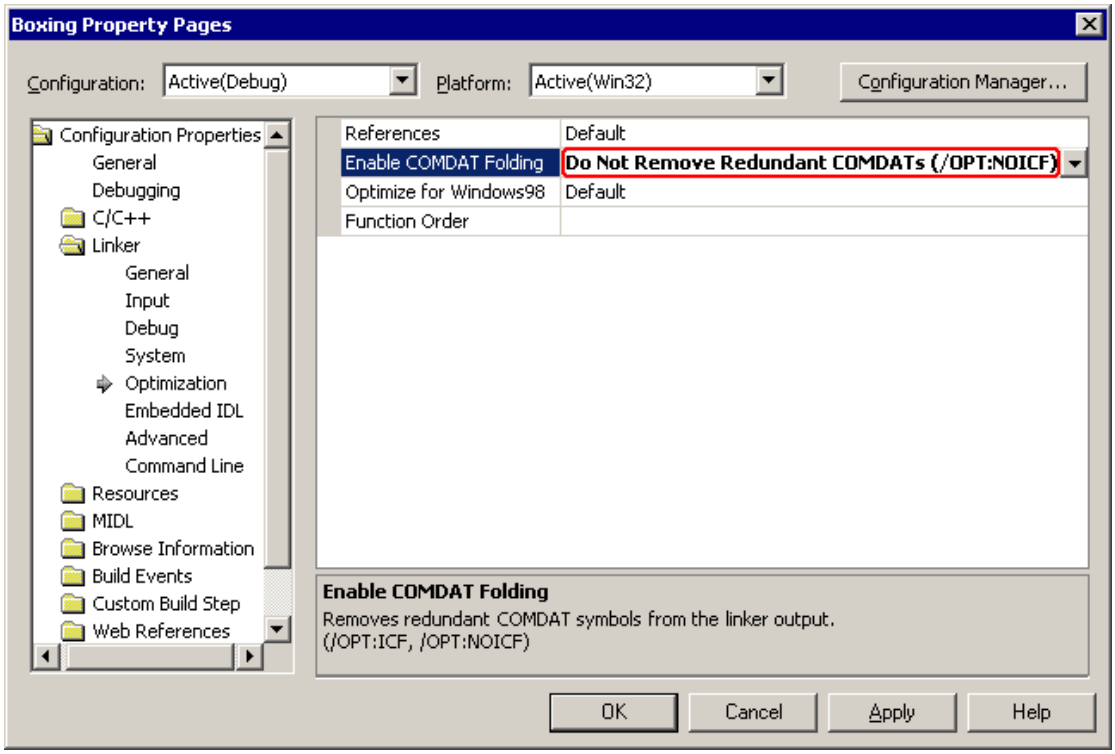
Profiling Duplicated Code

It is possible that two different routines, for instance, the AddRef methods of two different classes in a Visual C++ application, have the identical binary code (this is *duplicated code*). By default, the Visual C++ linker detects such routines and does not duplicate the code. Instead, it generates only one sequence of binary instructions that suits both occurrences. In this case, the debug information says that two different methods have the same address in memory. If both methods are included in profiling tasks, AQtime will profile only one of them. The other "duplicated" routines will be excluded from profiling (they will have zero profiling results). For all the routines whose code is duplicated, the **Analysis Result** column, which is available in the **Details** panel for the **Performance** profiler and in the **Report** panel for the Performance and **Coverage** profilers, will hold *Duplicated code*.

In other unmanaged applications, similar behavior of linkers regarding dublicated code is possible as well.

If your Visual C++ 7.x application includes routines that have duplicated code and you want to profile them as separate routines in AQtime, do the following:

- In Visual Studio .NET 2002, Visual Studio .NET 2003 or Visual Studio 2005, open the **Project Properties** dialog for the project you are going to profile (e.g. by using **Project | Properties**).
- Select **Configuration Properties | Linker | Optimization** on the left of the dialog.
- Locate the **Enable COMDAT Folding** option on the right of the dialog.
- Set this option to *Do Not Remove Redundant COMDATs (/OPT:NOICF)*.



- Click OK to close the dialog.
- Rebuild the project.

Now your project's debug information will have a individual section for each routine whose code is duplicated among other routines. As a result, these routines will not be excluded from profiling in AQtime. Instead, you will be able to profile these routines separately and get non-zero profiling results for each of them.

Profiling Small Functions

The Performance and Coverage profilers cannot profile functions that occupy less than *five bytes* in binary code. It concerns both managed and unmanaged code. This is a rare case. One single parameter (if used), or one single call to another function, will make the binary code larger than five bytes.

There is no override for this limit. The only workaround is to make the small functions larger, if you must absolutely profile them. You may turn off compiler optimizations that may be making your binary code smaller. Or you may simply add data to the function source to make it larger:

```
void LittleFunction()
{
```

```
  #ifdef profile
    int tmp[5];
    tmp[0] = 1; // prevents the compiler from discarding
                // the declaration
  #endif
...
}
```

If you choose this path, use conditional directives, as shown above, or remember to remove the added code after profiling.

Profiling Inline Functions

AQtime's profilers track entry and exit points of a function. When a function is set as inline, the compiler may insert a copy of the function body in each spot it is called (or it may disregard the directive). Obviously, with a true inlined function, there are no entry and exit points to track, so AQtime will include profiling results of that function into results of its parent function (or functions).

If you want an inline function to be profiled, you must set your compiler not to inline it. The manner, in which you can do this, depends on the compiler you use. Below is information for Microsoft Visual C++, Borland C++Builder and .NET compilers.

Microsoft Visual C# .NET, Visual Basic .NET and Other .NET Compilers

The JIT compiler compiles and optimizes the code of a managed application during the application run. The compiler uses specific algorithms to decide whether to inline a function or not. The easiest way to disable inlining for applications created with .NET compilers is to enable the *Disable inlining* option of the Performance or Coverage profiler before the profiler starts. When this option is on, routines in the profiled application are not inlined.

An alternative way to using the option is to specify the NoInlining method attribute in source code. For instance:

```
[C#]

using System.Runtime.CompilerServices;

  ...

[MethodImpl(MethodImplOptions.NoInlining)]
public void foo()
{
   // function code
   ...
}
```

Microsoft Visual C++ 6.0 and Visual C++ 7.x (Unmanaged Code)

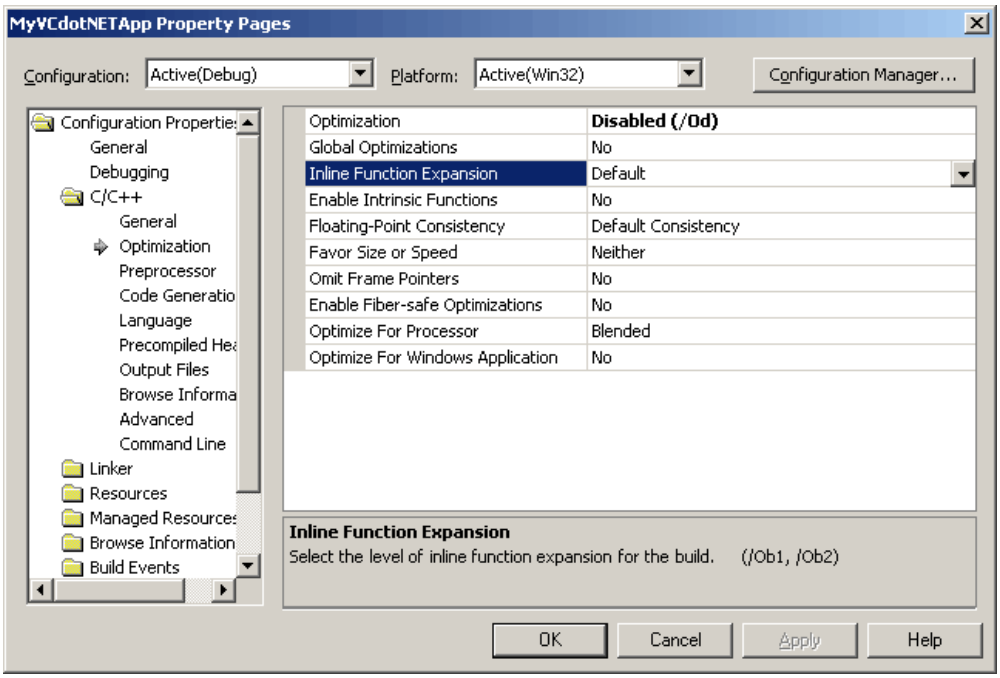
Both Microsoft Visual C++ 6 and Microsoft Visual C++ 7 distinguish several ways, in which an inline function is specified as inline:

- Using the `inline` or `__inline` keywords.
- For a member function, having its body declared in the class definition.

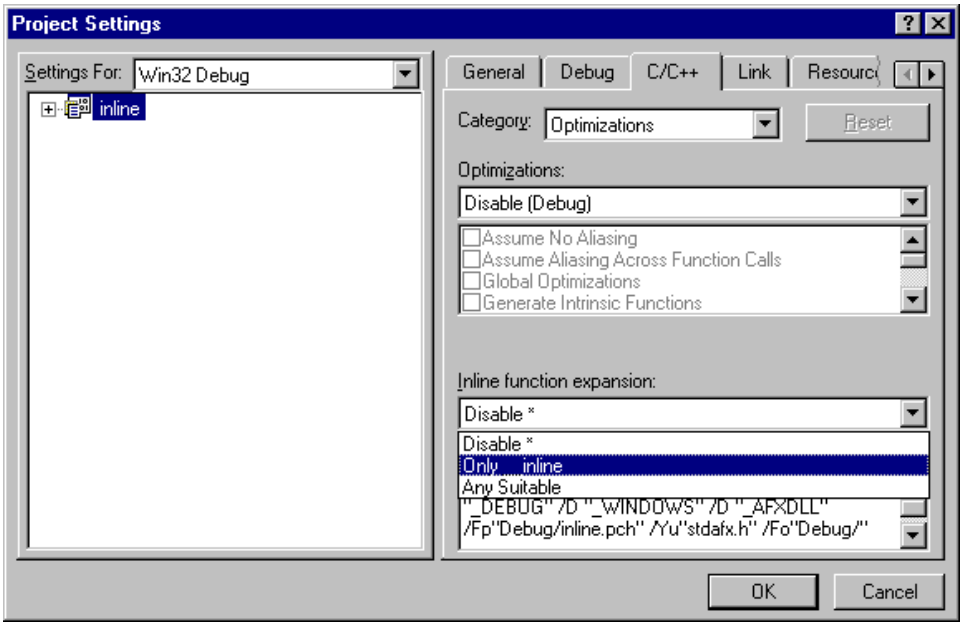
- Using `#pragma auto_inline` to tell the compiler to inline functions according to criteria of its own.

The Visual C++ compiler includes the *Inline Function Expansion* option that specifies which inline functions the compiler will inline. To modify this option:

- In Visual Studio .NET 2002, Visual Studio .NET 2003 or Visual Studio 2005:
  - Select **Project | <project name> Properties** from the main menu of Visual Studio. This will call the Project Properties dialog.
  - In the dialog, select the **Configuration Properties | C\C++ | Optimization** node from the tree on the left.



- In Visual Studio 6.0:
  - Select **Project | Settings** from the main menu of Visual Studio. This will call the **Project Settings** dialog.
  - In the dialog, switch to the **C\C++** tabbed page.
  - Select **Optimizations** in the **Category** box.



Possible values for the option are --

Default	Disables inlining. This is the simplest suitable solution for using AQtime.
Only __inline	The compiler will inline functions marked with the `inline` or `__inline` directive or member functions of a class. These functions will not be available for AQtime. All other functions will be available for profiling.
Any suitable	The compiler will produce inline code for all functions marked as inline as well as for any other suitable functions defined under `#pragma auto_line`. This is the setting most likely to cause problems, because it is not possible to predict which functions will be inlined and thus will be unavailable to AQtime.

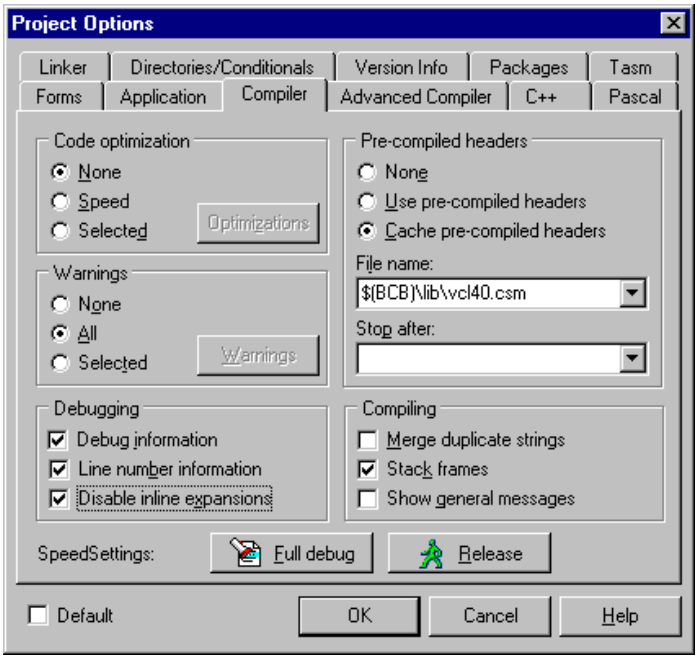
So, in order to make all the inline functions available for profiling, we recommend to set the Inline Function Expansion option to *Default*.

Borland C++Builder

Borland C++Builder includes the Disable inline expansions option that lets you disable inlining. That is, if this option is on, the compiler produces standard code for all inline functions, so AQtime can profile inline functions of any kind.

To modify this option:

- Select **Project | Options** from the C++Builder's main menu. This will call the **Project Options** dialog.
- Switch to the **Compiler** tabbed page. The *Disable inline expansions* option is in the **Debugging** section of this page.



Samples

To see how inline functions are profiled, you can use a sample application supplied with AQtime, *Inline*:

- <AQtime>\Samples\Unmanaged\Inline\VC - Microsoft Visual C++ (Visual Studio 7.x project)
- <AQtime>\Samples\Unmanaged\Inline\VC2005 - Microsoft Visual C++ (Visual Studio 2005 project)
- <AQtime>\Samples\Unmanaged\Inline\BCB - Borland C++Builder
- <AQtime>\Samples\Unmanaged\Inline\GCC - GCC

You can download compiled versions of these sample applications from http://www.automatedqa.com/downloads/aqtime/.

Profiling Template Functions

In C++ programs several different functions can be implemented from the same template. The debug info for these separate implementations will refer to the same source lines, those for the template definition. In other words, going by debug info, all these implementations are just one function. Thus, when using Coverage and Performance profilers, the grid of the Editor panel may display incorrect profiling information, for instance that a function was executed when it was not (only another from the same template was executed). The Report panel, however, gives the true profiling results. So, when analyzing profiler results for template functions please use the Report panel to view the correct results.

Profiling Routines That Hold Unsafe Code

Some of AQtime's profilers cannot profile routines that include unsafe code. Unsafe code means that your routine's binary code contains both binary code instructions and data (numeric and string constants, jump tables and so on). AQtime cannot isolate these two from each other. This may happen with some Delphi and Visual C++ routines that contain strings or ASSERTs and that you are going to profile at the line level. The fact is that the compiler places the data after the ret instruction, but debug information "includes" the data size into

the routine's code, so AQtime is unable to determine where the routine's binary code ends in memory. Below are examples of two routines that are possible candidates for having unsafe code:

```
[Visual C++]
void MyFunc()
{
  ...
  int cnt = objCollection->getCount();
  ASSERT(cnt < 0, "Invalid value.");
  ...
}
```

```
[Delphi]
procedure MyFunc()
begin
  ...
  MessageBox(0, 'Text', 'Caption', MB_OK);
  ...
end;
```

To solve the problem, you have to change the source code in order for the compiler to generate binary code in a different way. Here are some possible ways to do this:

- If you use the *Debug* configuration for your Visual C++ application, recompile it in the *Release* configuration.
- Divide your routine into a number of routines with smaller sizes. This will decrease the amount of data kept within the routine's binary code.
- Try to decrease the amount of data kept within the routine's code by moving string constants outside the routine. For instance, if your routine includes a call to the `MessageBox` function:

  ```
  [Delphi]
  procedure MyFunc()
  begin
    ...
    MessageBox(0, 'Text', 'Caption', MB_OK);
    ...
  end;
  ```

  You can move the string constants ('Text' and 'Caption') outside the routine:

  ```
  [Delphi]
  string cText = 'Text';
  string cCaption = 'Caption';

  procedure MyFunc()
  begin
    ...
    MessageBox(0, cText, cCaption, MB_OK);
    ...
  end;
  ```

- Create a stub routine that will call your routine and profile this stub routine at line level:

  ```
  [Visual C++]
  ```

```
#pragma optimize("", off)
void StubFunc(int someValue)
{
  MyFunc(someValue);
}
#pragma optimize("", on)
```

We used the `#pragma optimize` directives to make certain that `StubFunc` occupies more than five bytes in memory. This is a rare case, however. If a routine uses at least one parameter, or if it calls another function, its binary code will be larger than five bytes.

Profiling Routines That Do Not Have the ret Instruction

AQtime profilers cannot correctly analyze functions that exit without the `ret` instruction, but through a jump. You may choose to put such routines into an excluding area. But if you wish to profile them, all you have to do is to make a few modifications in source code. The easiest way is to add an assembler block to the end of the routine. For instance:

```
[Visual C++]
...
__asm{
    jmp tmp
    ret
  tmp:
}
```

```
[Delphi]
...
asm
  jmp tmp
  ret
 tmp:
end;
```

If this does not help, try modifying the function's code. For instance, Borland Delphi uses a .dpr file that holds code used to initialize an application, display the splash screen, load any data common to the entire application etc. This code is placed between `begin` and `end`, without a function name, as per Pascal rules. Call this the *main procedure* for the application.

The *main procedure* does not exit normally; it simply ends when `Application.Terminate` is executed. Therefore, the Performance or Coverage profiler, for instance, cannot profile it. If you wish for it to be profiled, you simply have to move its code to an ordinary procedure, with a name, and call that from the .dpr file.

Suppose, the dpr file originally used the following code:

```
[Delphi]
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

 {$R *.RES}
```

```
begin
  { a custom procedure that loads data common for
    the entire application }
  LoadCommonData;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end;
```

The trick will be to add the `DoMain` procedure to Unit1.pas (including an `interface` part declaration), cut and paste the code of the *main procedure* and replace it with the call to `DoMain`, leaving the dpr code as below:

```
[Delphi]
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

  {$R *.RES}

begin
  DoMain;
end;
```

Unit1.pas is changed on this model:

```
[Delphi]
interface
 ...

  procedure DoMain;

implementation

procedure DoMain;
begin
  { a custom procedure that loads data common for
    the entire application }
  LoadCommonData;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end;
```

The Performance and Coverage profilers will now be able to profile `DoMain`.

Profiling Child Routines Along With Parents

This topic explains precautions to take when profiling a function that calls other functions (child functions).

Let's call the caller function `ParentFunction`. If --

- you are not using *Full Check* or *Profile Entire .NET Code* (if you profile a managed application),

- `ParentFunction` is included in a checked profiling area,
- but some or all of the child functions it calls are included in no checked area,

then this will happen:

The child functions will not be profiled, and therefore will not appear in the Details panel for `ParentFunction`. Their results (for instance, the execution time), will still be counted, but as part of `ParentFunction`'s results (for example, the execution time of the `ParentFunction`'s own code). AQtime is correctly noting `ParentFunction`'s entry and exit times, but "knows" nothing about the time spent "outside" on calls to unprofiled child functions. This is can easily lead you to assume the slow code is in `ParentFunction`, when actually it is in some unprofiled child function.

So, when you are worried about a function's execution time (or other results), make sure that its children are profiled along with it. You may identify the child calls by looking up the parent function in the Editor panel. This is also a good occasion to think of triggers.

Profiling Startup Code

The actions you need to perform to profile the startup code of an application depends on whether the process is started by AQtime or by some other process or the operating system.

Regardless of the way your application starts, keep the following in mind:

- Make sure the startup code is included in profiling areas in the Setup panel.
- The startup code may not contain the `ret` instruction (this happens, for instance, with code of Delphi's .DPR files), so to profile this code you may need to place it to the routine specially created for this. For more information, see *Profiling Routines That Do Not Have the ret Instruction*.

If the process is started under AQtime, the profiling engine traces the execution of all the application's functions added to profiling tasks, so profiling the startup code is not a problem. You just need to add it to an including profiling area (or areas).

If your process is not started by AQtime, but started by another process or by the operating system, then to profile it, you could attach AQtime to it (see *Attach to Process*). However, attaching requires time, so most likely you will not be able to profile the startup code of your process.

Specifying the host process as a Host Application for the desired process in the Run Parameters dialog will not solve the problem as well. If you do so, AQtime will only profile the code that is executed within the address space of the host process, not the desired process. In other words, this approach works for dynamic link libraries or in-process COM servers, but does not work for applications.

To solve the problem, you can do any of the following:

- Profile your application in COM Server profiling mode (you can do this even if the process to be profiled is not a COM server).
- Modify the registry settings so that when the operating system gets a command to start a process, it automatically launches AQtime, which you can then use to profile your application.

Using COM Server Mode

When you select the COM Server profiling mode for the application, then AQtime automatically waits until the application's executable is loaded in memory and then starts profiling the application code. COM Server mode was designed for profiling startup code of COM servers that are launched by the operating system. However, you can use this mode for profiling other applications, even if they are not COM servers:

- Open your project in AQtime and add the startup code routines into profiling areas (see *Controlling What to Profile*).
  - Select **COM Server** from the **Profiling Mode** dropdown list box that is displayed on AQtime's Standard toolbar (if you use AQtime integrated into Microsoft Visual Studio, you can choose the COM Server mode from **AQtime** toolbar. If you use AQtime integrated into Borland Developer Studio's, you can choose this mode from the **Run.AQtime** toolbar).
  - Select **Run | Parameters** from AQtime's main menu. This will open the **Run Parameters Dialog (for COM Server Mode)**. (To invoke the dialog in Visual Studio, select **Profile | Parameters** from Visual Studio's menu. In Borland Developer Studio, click **Parameters** on the **Run.AQtime** toolbar).

    In the dialog:
    - In the **Client Application** box, specify the fully qualified name of the executable that will launch your application.
    - Press **OK** to close the dialog.
  - Press **Run** to start profiling. (In Microsoft Visual Studio, select **Profile | Run** or **Debug | Run** while one of AQtime panels is active. In Borland Developer Studio, select **Run | Run With Profiling**).

  AQtime will display a message informing you that the client application should be run.
  - Perform actions that will lead to launching your application.
  - Work with your application as needed.
  - Close your application and the application that was specified in the Client Application box of the Run Parameters dialog.

Modifying Registry Settings

To modify the registry settings:

- Launch the Registry Editor (regedit).
- Open the **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options** node.
- Add a key for your application to this node:
  - Right-click the node and choose **New | Key** from the context menu. This will create a new key node under the Image File Execution Options node.
  - Right-click the new node and select **Rename** from the context menu.

    Change the key name to the name of your executable (for instance, *MyApplication.exe*). Only specify the file name and extension. The path is not needed.
- Specify AQtime as a debugger for the application. To do this:
  - Right-click the new key node and select **New | String Value** from the context menu. This will append a new string value to the created key.
  - Switch to the right panel of the Registry Editor, right-click the string value's node and choose **Rename** from the context menu.

    In the ensuing in-place editor, change the value name to **debugger**.
  - Right-click the value node again and select **Modify** from the context menu. This will call the **Edit String** dialog.

© 2010 AutomatedQA Corp.

- In the **Value** data box of the dialog, enter the fully qualified name of AQtime.exe, for instance, *C:\Program Files\Automated QA\AQtime 5\Bin\AQtime.exe*.
- Click **OK** to save the changes and to close the dialog.

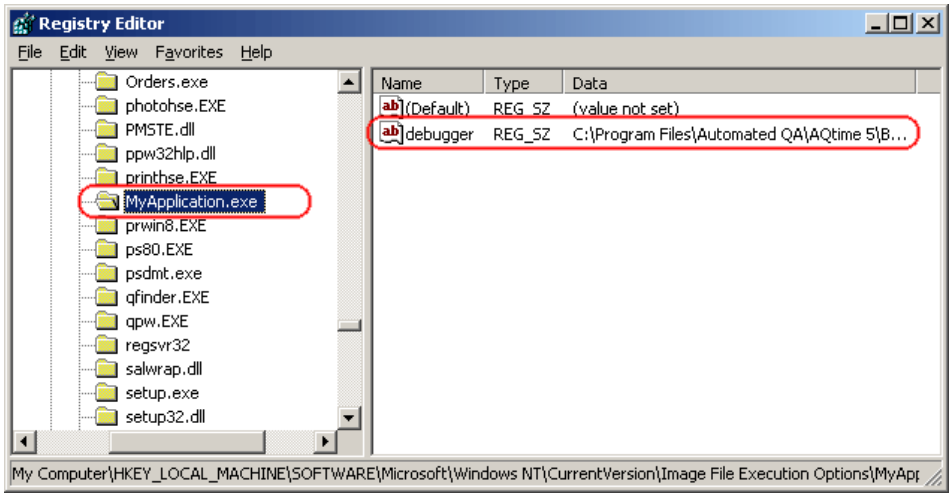
Now the operating system will launch AQtime every time your application starts. The general profiling procedure is:

5. Run the application that will launch your application.
6. Modify the registry settings as it was described above.
7. Perform the actions that will lead to launching your application. The operating system will start AQtime as the registry settings specify it as a debugger for your application.
8. In AQtime, specify the profiling areas and triggers, start profiling and profile your application as you normally would.
9. Close your application and explore profiling results in AQtime.

Do not forget to remove the key from the Registry when the automatic launch of AQtime is no longer needed.

If you need to enable or disable the automatic launch frequently, you can create two .reg files:

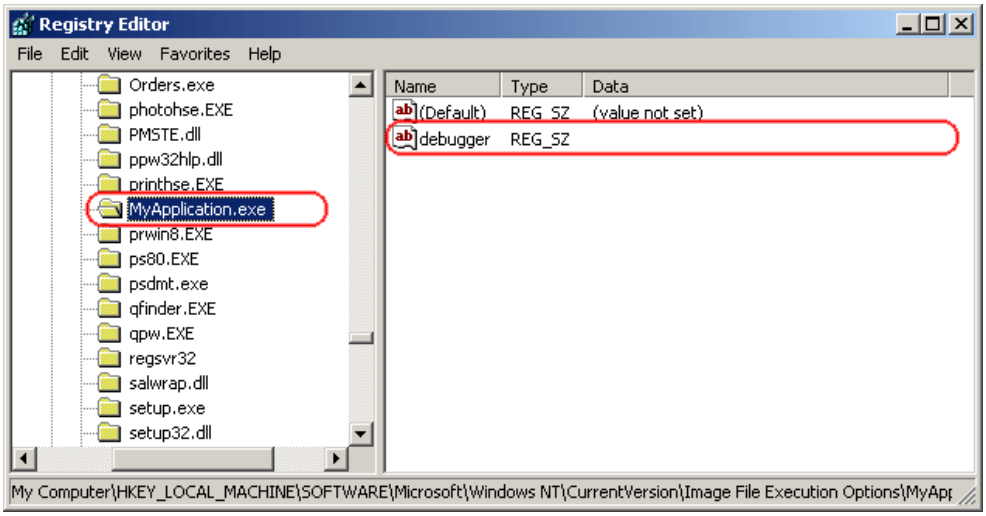
- One of the files will include the Registry key and the debugger value with the path to AQtime:



To create this file:

1. Modify the registry as it was described above.
2. Select the key in the tree view on the left of the Registry Editor.
3. Choose **File | Export** from the editor's main menu and specify the file name in the ensuing **Export Registry File** dialog.

- Another file will store the Registry key and the debugger value, but it will not store the path to AQtime:



To create this file:

1. Modify the registry as it was described above.
2. Clear the data of the **debugger** value.
3. Export the key to a file by choosing **File | Export** from the main menu of the Registry Editor.

Now, when you need to enable or disable the automatic launch of AQtime for your application, you can execute one of the .reg files.

Controlling Profiling From Application Code

AQtime lets your applications handle the profiling process. This has the same effect as using the **Enable/Disable Profiling** toolbar and menu item. Note that these commands only work if there is only one instance of AQtime running.

This is useful when areas plus triggers and actions do not give you the control you seek, or where they would, but at the cost of some complications, or simply where you want to set triggers or actions from source code rather than from the AQtime user interface.

All the functions that you need to do this are in a single file that you add to your source files. Once you have it linked, turning profiling on and off is a matter of a few simple calls.

If you use…	Add the following file
Microsoft Visual C# .NET	<AQtime>\SDK\CS\AQtimeHelpers.cs
Microsoft Visual Basic .NET	<AQtime>\SDK\VBNET\AQtimeHelpers.vb
Microsoft Visual C++, Borland C++Builder, Borland C++ or Intel C++	<AQtime>\SDK\CPP\AQtimeHelpers.cpp
Microsoft Visual Basic	<AQtime>\SDK\VB\AQtimeHelpers.bas
Borland Delphi	<AQtime>\SDK\Delphi\AQtimeHelpers.pas

Each of these files includes the `EnableProfiling` function. It enables or disables profiling according to its Boolean parameter.

You can try it with the sample application, *OnOffProfiling*, that is part of the AQtime installation (in source). Alternatively, you can try the following sample code in an application of your own. In either case, before running the test set up the application in AQtime so that the application has full control over profiling:

- Select *Full Check* and/or *Profile Entire .NET Code* in AQtime's Setup panel. For more convenient result analysis, you should uncheck all profiling areas, except *Full Check* and *Profile Entire .NET Code*.
- Be sure the **Enable/Disable Profiling** button is not pressed so AQtime does not begin profiling the application by itself.

Sample code:

```
[Visual C++]
 . . .
// Enables profiling
EnableProfiling(true);

// Calls a function
My_Function(Param1, Param2);

// Disables profiling
EnableProfiling(false);
```

Note that if your application already contains a routine with the name `EnableProfiling`, you may need to use the `AQtimeHelpers` namespace or the `AQtimeHelpers` unit name (this depends on your compiler) when calling the `EnableProfiling` routine, for example:

```
[Visual C++]
 . . .
// Calls EnablesProfiling using the namespace
AQtimeHelpers::EnableProfiling(true);
```

```
[Delphi]
...
 // Calls EnableProfiling using the unit name
AQtimeHelpers.EnableProfiling(True);
```

AQtime help file includes a tutorial that explains how you can control profiling from your application: *Enable/Disable Profiling Tutorial*. Be sure to read it to gain a better understanding of how it works.

Generating Dumps for Profiled Applications

When profiling your application in AQtime, you may command AQtime to generate a dump for the profiled application. The generated file will contain information about the application's memory, call stacks, loaded modules and CPU registers' values. This information may help you understand what happens within the application at any given point in time. For instance, if your application hangs, you may generate the error report and use its data to find the cause of the problem.

To generate an error report:

- Click **Generate Process Dump** on the Event View panel's toolbar.

  AQtime invokes the save file dialog.
- In the dialog, specify the location and name of the dump file and press **Save**.

  AQtime will save the error report to the specified file and close the dialog.
- Then you can either continue profiling or terminate the profiled application.

- To analyze the data of the dump file, open the file in AutomatedQA AQtrace or in Microsoft Visual Studio 2005 or 2008. For more information on how to analyze dumps, see the AQtrace or Visual Studio documentation (in AQtrace, dump files are called *error reports*).

Notes:

- The format of the dump file generated by AQtime differs from the format of error report files adopted in AQtrace. So, when opening the generated dump file in AQtrace the latter will display a message box informing you that the file was generated by another tool, but not AQtrace.
- Currently, the generated dump file contains the "native-code" call stack. If you generated a dump for a .NET application, the dump will contain the native-code entries, but not the names of managed routines.

You can also command AQtime to **generate dumps automatically** when an exception occurs in the profiled process. To do this, enable the *Generate dump on exception* setting of the Event View panel and specify the folder that the dumps will be saved in, in the *Dump folder* setting. Note that the folder must exist; AQtime does not create it automatically.

When an exception occurs in the profiled application, AQtime checks the value of the *Generate dump on exception* setting. If the setting is enabled, AQtime automatically generates a dump for the exception, saves the dump file to the folder, specified by the *Dump folder* setting and adds an informative message to the Event View panel.

The name of the dump file has the format `ProjectName_nn.dmp`. *ProjectName* is the name of your AQtime project and *nn* is a number (1, 2, 3 and so on).

Notes:

- Dumps are not generated for those exceptions that are filtered out and not logged in the Event View panel (see *Exceptions in the Event View Panel*).
- When a .NET exception occurs, the CLR generates the appropriate system exception. The dump file contain the call stack for this system exception. It does not contain the call stack for the .NET exception.

Assigning Names to Threads

AQtime gathers profiling results per application thread (see *Profiling Multiple Threads*). To identify a thread in profiling results, AQtime typically uses the thread identifier. For instance, you may see the names like *Thread #256*, *Thread #541* in the Explorer panel. The analysis would be easier if a thread had a more descriptive name. For instance, if you are looking for results of the main application thread, then the name *Main Thread* is much more useful than *Thread #256*.

For .NET applications running under the .NET Framework version 2.0 or later, AQtime is able to obtain the user-defined name of a CLR thread assigned through the `Thread.Name` property. So, using this property you may assign desired names to the CLR threads created in your .NET applications running under the .NET Framework ver. 2.0 (or later).

To assign descriptive names to threads in other applications, use the `SetCurrentThreadName` function that is defined in the AQtimeHelpers file that is included into the AQtime package (see below). The name assigned to a thread with this function will be used instead of the default thread name in profiling results.

The use of descriptive names for threads will not only simplify the result analysis, but will also improve the merging of results: AQtime will automatically merge the results for those threads that have the same name. Other threads will be included into the merged result set as separate items. For more information about this, see *Comparing and Merging Results*.

Note: If you assign custom names to threads for comparison and merging purposes, do not use the names that start with *Win32 Thread*, *CLR Thread* or *COM Thread*. AQtime considers such names as default ones and it will not detect a thread as renamed when they are used.

To assign a name to a thread, follow these steps:

- Open your application's project in the development tool you use.
- Include the AQtimeHelpers file into your application project. This file is located in the following folder:

If you use…	Add the following file
Microsoft Visual C# .NET	<AQtime>\SDK\CS\AQtimeHelpers.cs
Microsoft Visual Basic .NET	<AQtime>\SDK\VBNET\AQtimeHelpers.vb
Microsoft Visual C++, Borland C++Builder, Borland C++ or Intel C++	<AQtime>\SDK\CPP\AQtimeHelpers.cpp
Microsoft Visual Basic	<AQtime>\SDK\VB\AQtimeHelpers.bas
Borland Delphi	<AQtime>\SDK\Delphi\AQtimeHelpers.pas

  The file holds the declaration of the **`SetCurrentThreadName`** function. This function assigns the name to the thread where it is called. It uses the only parameter - the name of a thread to set.

- Call the `SetCurrentThreadName(`*`Name`*`)` function in your code (we recommend that you call this routine at the beginning of your thread function). If a thread already has a name, it will be renamed.

```
[Visual C++]

SetCurrentThreamName("Custom thread name");
```

  If your application already contains a routine with the name SetThreadName, you may need to use the AQtimeHelpers namespace or the AQtimeHelpers unit name (this depends on your compiler) when calling the `SetCurrentThreadName` routine:

```
[Visual C++]
 . . .
// Calls SetThreadName using the namespace
AQtimeHelpers::SetCurrentThreadName("My thread name");
```

```
[Delphi]
 . . .
// Calls SetThreadName using the unit name
AQtimeHelpers.SetCurrentThreadName("My thread name");
```

- To obtain the name assigned to the current thread, call `GetCurrentThreadName` (like `SetThreadName` this function is declared in the AQtimeHelpers file).

The OnOffPofiling sample application shipped with AQtime contains code that demonstrates how you can assign descriptive names to threads in your application:

- <AQtime>\Samples\Unmanaged\OnOffProfiling\VC - Microsoft Visual C++ (Visual Studio 7.x project)

- <AQtime>\Samples\Unmanaged\OnOffProfiling\VC2005 - Microsoft Visual C++ (Visual Studio 2005 project)
- <AQtime>\Samples\Unmanaged\OnOffProfiling\Delphi - Borland Delphi
- <AQtime>\Samples\Unmanaged\OnOffProfiling\BCB - Borland C++Builder

You can download compiled versions of these sample applications from http://www.automatedqa.com/downloads/aqtime/.

Specifying Path to Debug Info Files

Different compilers generate different types of debug information. For instance, a Visual C++ compiler generates debug info in the PDB format and Delphi generates debug info in the TD32 format.

Depending on its type and the compiler settings, debug information can be compiled within an executable, or it can be generated as a separate file (or files) that reside(s) at specific location relative to the executable. This typically happens with debug information of the PDB format. The compiler places the .pdb file in a folder, where the executable is located. However, in general, the debug info file can reside in any other folder or even on another computer on a local network or on the Internet.

Some developers use this feature to save disk size needed for debugging applications on several computers. When debugging these applications, the developers place debug info files in a folder and share this folder on the local network or on the Web. The debugging tools that are running on other computers can access this shared folder and obtain the debug information for the modules being debugged. This approach saves the overall amount of disk space occupied by the application's files and modules, since the debug info files are stored in a shared location and are not copied to each workstation.

When you add an executable or DLL to your AQtime project, AQtime searches for the debug info file (or files) in the default location, that is, in the folder where the compiler typically generates the debug info file. If the debug info file resides in another location, AQtime will not be able to find it and will report that the executable has no debug information.

AQtime includes the Symbols Options dialog where you can specify the location for the debug info files. AQtime will search for the debug info files in the folders that are specified in the dialog.

In order for AQtime to be able to use this functionality, SymSrv.dll must be installed on your computer. This DLL is deployed as part of Microsoft's Debugging Tools for the Windows package. You can download this package from Microsoft web site.

To specify the folder for the search:

- Open the Symbols Options dialog. To do this:

  If you use AQtime stand-alone:

  - Select **Options | Options** from AQtime's main menu. This will open the Options dialog.
  - Choose the **Services | Symbols** group in the tree view on the left of the Options dialog.

  If you use AQtime integrated into Microsoft Visual Studio:

  - Select **Tools | Options** from the main menu of Visual Studio. This will invoke the Options dialog.
  - Choose the **AQtime | Services | Symbols** group in the dialog.

  If you use AQtime integrated into Borland Developer Studio:

  - Select **Profile | Options** from the main menu of Borland Developer Studio. This will invoke the Options dialog.

  - Choose the **Services | Symbols** group in the tree view on the left of the Options dialog.
- In the **Symbols Path** section of the dialog, press **Add**. AQtime will append a new blank line to the **Symbols Path** list and activate an in-place editor.
- Specify the desired folder in the in-place editor. You can type the folder name or press the ellipsis button and choose the folder in the ensuing standard Browse for Folder dialog. Specify the fully-qualified folder name, for instance, *C:\MySources\Symbols* or *//MyServer/Symbols*. Project-relative paths are not supported.

  You can also enter the URL (web folder) that holds the desired debug info file (for instance, *http://msdl.microsoft.com/download/symbols*).
- Make sure the check box for the added line is selected. AQtime will search for the files only in those folders (or URLs), whose check box is selected in the dialog.
- Press **Move Up** or **Move Down** to set the desired position of the line among other search paths (AQtime will search for the files starting with the folders that are specified at the top of the list. If the file is found, the other folders are not checked).
- Press **OK** to save the changes.

If the debug info files are available via the Internet, then AQtime will download them to your computer and save them to a temporary folder. The file will reside in this folder until you close AQtime. When you close it, the files will be deleted. So, next time you launch AQtime, it will download the files again. To avoid this repetitive downloading, you can specify the folder, in which the downloaded files will be stored, in the **Cache symbols directory** edit box. If you specify a folder, AQtime will download the files to this folder and will use these files during the next sessions. The files will remain in the folder until you remove them manually.

Profiling .NET Applications – Peculiarities

User account:Running NET applications underProfiling of .NET applications has the following peculiarities:

- **Profiling .NET applications via the network**

  Due to certain security reasons, AQtime cannot profile those .NET applications that reside on another computer. Profiling of these applications causes an exception that occurs within the application code due to security peculiarities of the .NET Framework. To profile these applications, launch AQtime on the computer where the application is located.
- **Profiling .NET applications running under another account**

  If you install AQtime for a single user account, you will not be able to profile .NET applications that run under another user account.

Note: Both of these peculiarities concern profilers that launch your application. They are not applied to profilers that analyze your code statically.

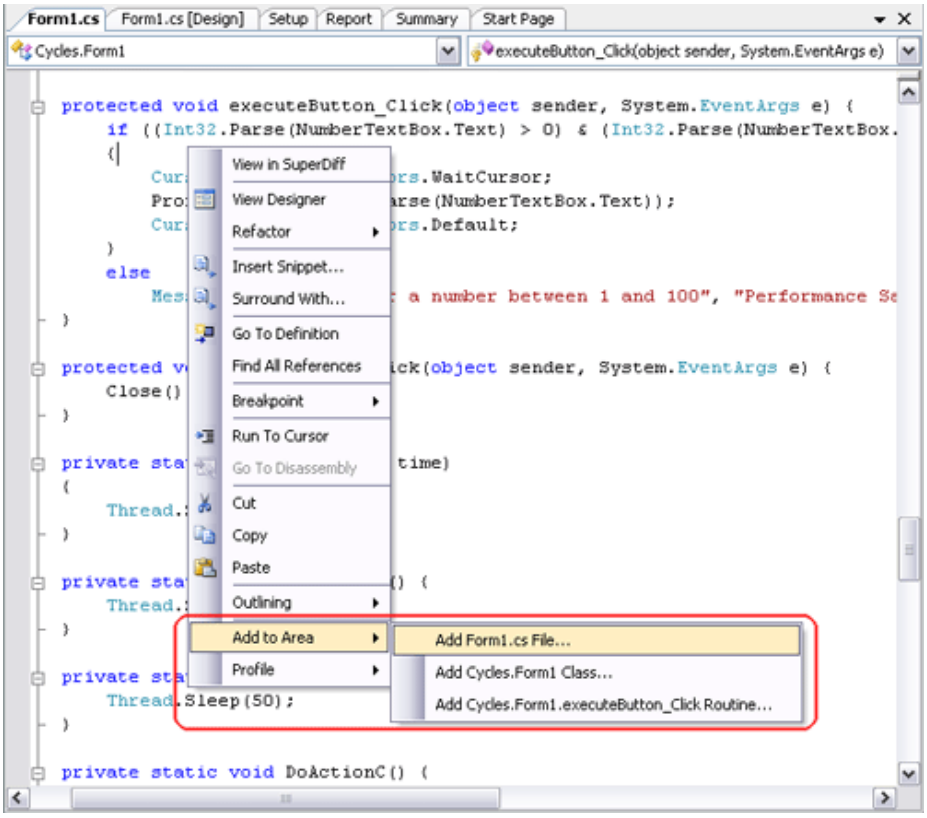
- **Profiling console .NET applications**

  AQtime does not trace exceptions that occur in console applications created for .NET Framework ver. 1.0 and 1.1.

Adding Code to Profiling Areas From Code Editor

To specify the files, classes and routines to be profiled, you create one or several profiling areas in the Setup panel, add the desired files, classes or routines to these profiling areas, select the areas for profiling and then start profiling.

When working in Microsoft Visual Studio, you can also add routines, classes and files to the desired profiling area directly from Visual Studio's Code Editor. To do this, right-click your code in the Code Editor and use the items of the Add to Area submenu:



The Add to Area item is available only if your Visual Studio solution contains an AQtime project.

The **Add to Area** item opens a submenu that contains three items:

- Add *file_name* File
- Add *class_name* Class
- Add *routine_name* Routine

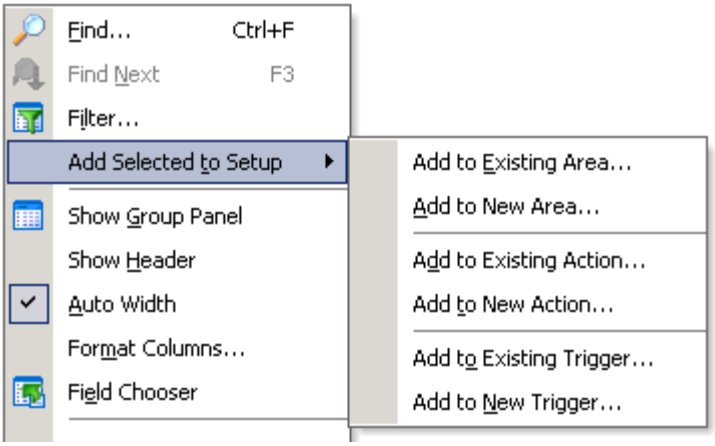
Upon selecting any of these menu items, AQtime displays the Select Area dialog, in which you specify the area to which the routine, class or file will be added.

Working With Profiler Results

Adding Selected Routines and Classes to the Setup Panel

You can add routines or classes to profiling areas, triggers and actions directly from the Report panel. To do so, select one or more rows in the Report panel and select **Add Selected to Setup** from the context menu. This will open another menu containing six items:

© 2010 AutomatedQA Corp.



The **Add to Existing Area**, **Add to Existing Action** and **Add to Existing Trigger** commands display the Select Area, Select Action and Select Trigger dialogs respectively. In these dialogs, you can select an existing profiling area, action or trigger and add the selected routines or classes to it.

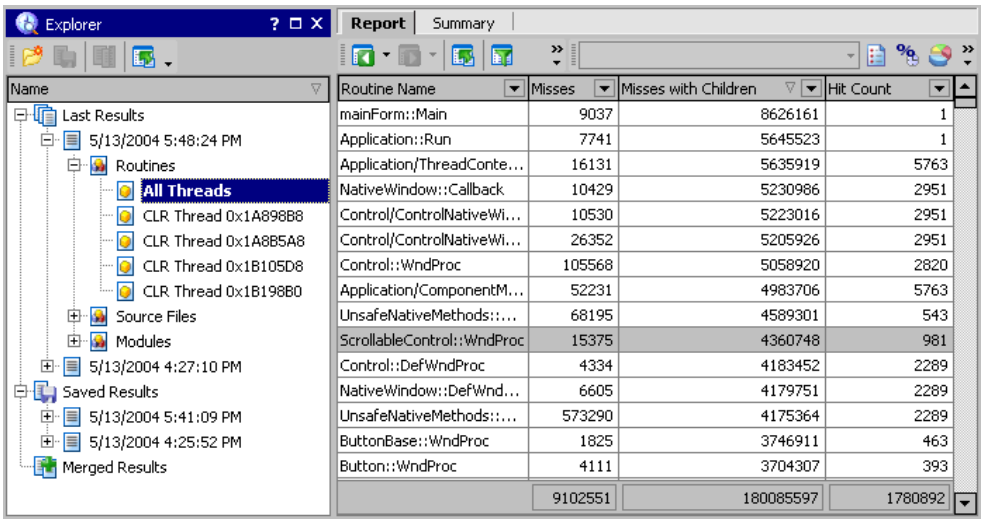
The **Add to New Area**, **Add to New Action** and **Add to New Trigger** commands display the Add Area, Add Action and Add Trigger dialogs where you can create a new area, action or trigger and then add the selected routines or classes to it.

These features let you easily separate the desired routines from other application functions and then profile the desired routines only. Suppose you run the Performance profiler against the entire application; you found several slow routines and want to profile them. In this case, you can select these routines in the Report panel, add them to a new profiling area via the Add to New Area command, uncheck other profiling areas in the Setup panel and then start a new profiler run. Creating a new profiling area from the Report panel is faster than creating a new area and adding the desired routines to it in the Setup panel.

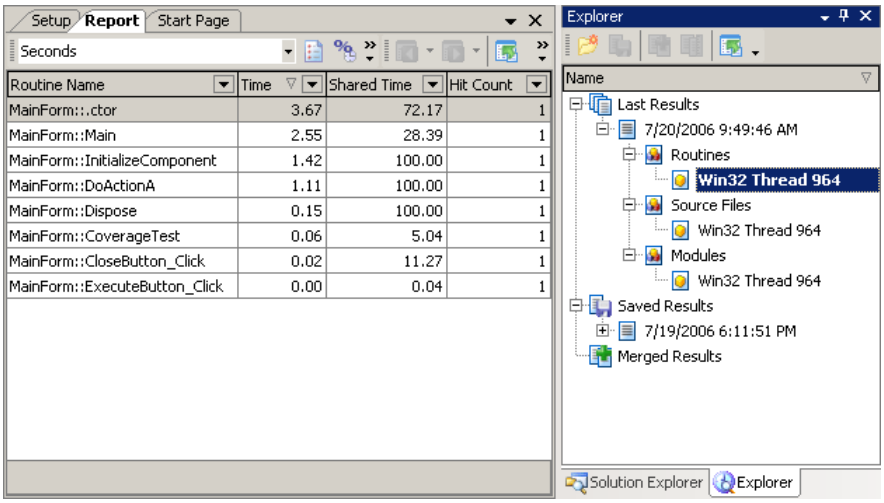
Note: The above commands are applicable to classes, if the items displayed on each row of the Report panel are classes (for example, the results of the Allocation profiler), as opposed to routines.

Comparing and Merging Results

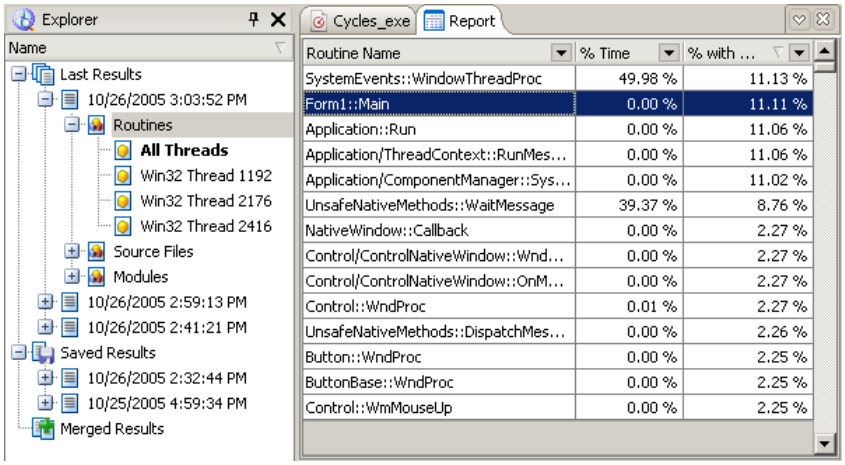
Within the Explorer panel, AQtime stores a “Last Results” archive of the most recent result sets (five sets by default). These are labeled with date and time, and you can add your description directly on screen. While a result set is still archived, you can choose to copy it to your own archive, Saved Results, where it will remain until you delete the set or delete the project from disk. All of the Explorer contents are specific to the current project and current profiler. See *Explorer Panel* for details on all these points.


Explorer
Report
Summary
Name
Last Results
5/13/2004 5:48:24 PM
Routines
All Threads
CLR Thread 0x1A898B8
CLR Thread 0x1A8B5A8
CLR Thread 0x1B105D8
CLR Thread 0x1B198B0
Source Files
Modules
5/13/2004 4:27:10 PM
Saved Results
5/13/2004 5:41:09 PM
5/13/2004 4:25:52 PM
Merged Results
Routine Name | Misses | Misses with Children | Hit Count
mainForm::Main | 9037 | 8626161 | 1
Application::Run | 7741 | 5645523 | 1
Application/ThreadConte... | 16131 | 5635919 | 5763
NativeWindow::Callback | 10429 | 5230986 | 2951
Control/ControlNativeWi... | 10530 | 5223016 | 2951
Control/ControlNativeWi... | 26352 | 5205926 | 2951
Control::WndProc | 105568 | 5058920 | 2820
Application/ComponentM... | 52231 | 4983706 | 5763
UnsafeNativeMethods::... | 68195 | 4589301 | 543
ScrollableControl::WndProc | 15375 | 4360748 | 981
Control::DefWndProc | 4334 | 4183452 | 2289
NativeWindow::DefWnd... | 6605 | 4179751 | 2289
UnsafeNativeMethods::... | 573290 | 4175364 | 2289
ButtonBase::WndProc | 1825 | 3746911 | 463
Button::WndProc | 4111 | 3704307 | 393
9102551 | 180085597 | 1780892

Explorer panel in AQtime standalone



Explorer panel in AQtime integrated into Visual Studio


Explorer
Name
Last Results
10/26/2005 3:03:52 PM
Routines
All Threads
Win32 Thread 1192
Win32 Thread 2176
Win32 Thread 2416
Source Files
Modules
10/26/2005 2:59:13 PM
10/26/2005 2:41:21 PM
Saved Results
10/26/2005 2:32:44 PM
10/25/2005 4:59:34 PM
Merged Results
Cycles_exe
Report
Routine Name | % Time | % with ...
SystemEvents::WindowThreadProc | 49.98 % | 11.13 %
Form1::Main | 0.00 % | 11.11 %
Application::Run | 0.00 % | 11.06 %
Application/ThreadContext::RunMes... | 0.00 % | 11.06 %
Application/ComponentManager::Sys... | 0.00 % | 11.02 %
UnsafeNativeMethods::WaitMessage | 39.37 % | 8.76 %
NativeWindow::Callback | 0.00 % | 2.27 %
Control/ControlNativeWindow::Wnd... | 0.00 % | 2.27 %
Control/ControlNativeWindow::OnM... | 0.00 % | 2.27 %
Control::WndProc | 0.01 % | 2.27 %
UnsafeNativeMethods::DispatchMes... | 0.00 % | 2.26 %
Button::WndProc | 0.00 % | 2.25 %
ButtonBase::WndProc | 0.00 % | 2.25 %
Control::WmMouseUp | 0.00 % | 2.25 %

Explorer panel in AQtime integrated into Borland Developer Studio

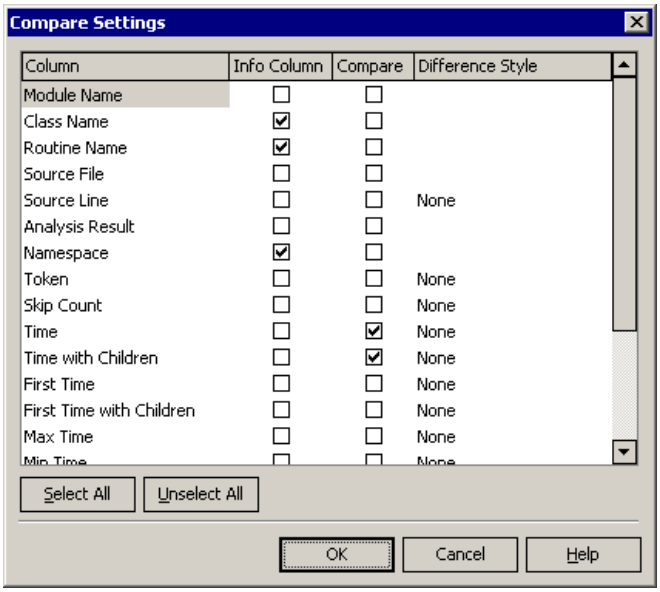
Besides allowing easy reference to past results, this system lets you set up **comparisons** between result sets or **merge** them into a new, combined result set.

Comparing Results

Suppose you have profiled a sorting procedure and discovered that it is slow. You may decide that the algorithm must be optimized. You will try something, and then profile it again. At this point the "Compare" facility steps in and lets you focus on the resulting differences in a single comparison report, laid out as a normal result report would be.

We will call each stored result set (that is, a child node of the dated nodes in the Explorer panel) a *record*. You can multi-select any number of records in the Explorer panel then choose **Compare** from the Explorer toolbar or from the context menu and, voila!, the comparative report will appear in the Report panel.

The comparison is configurable, of course. Select **Compare Settings** from the Explorer context menu and you will get the Compare Settings dialog:



The actual dialog depends on the current profiler. The checkboxes in the **Compare** column select the columns you want to show in your comparison report.

For numeric results, if you have selected only two records to compare, you can select a *Difference Style*. If you have more than two records, the **Difference Style** is *None*, which means that columns from each record will simply be shown side by side. Other Difference Styles are simply ways to "compact" the columns from the two records into one by doing a simple arithmetic operation on them to show the difference. These Styles use "Record 1" for the first record you selected, "Record 2" for the second.

Explorer options include an *Always set up Compare parameters* checkbox. If this is checked, then the Compare Settings dialog will pop up whenever you ask for a Compare.

Although the Details, Call Graph and Call Tree panels are not available in comparison mode, the Editor and Disassembler panels still work and display the most recent code for the routine selected in the Report panel.

Note that you can compare results from the same categories only. For instance, you cannot compare the Allocation profiler results, if one of them is stored to the *Classes* category and the other to the *Objects* category. If you compare the Performance profiler results, be sure they were generated by the same counter. AQtime cannot compare results that were generated by different counters.

Merging Results

Merging results means bringing them together into a new result set, as if it was another profiler result, except that the numeric fields are replaced by the sum, average, maximum or minimum of the values in the merged records (see more below).

The resulting record goes into the **Merged Results** section of the Explorer panel. Note that the Explorer shows it as if it held its source records as well, but this is only a way to identify the source. Only the merged results are kept in Merged Results. Like Current Results, Merged Results must be saved (from the toolbar or the context menu) to be kept beyond the auto-archive limit (default five records).

The advantage of merges is that they focus on important statistics for the collection of records selected, such as average results over several runs. The limitation is that the application must not have changed in ways important to profiler results. If function names have changed, for instance, then merging becomes pointless. The merging also becomes useless if a profiled function has been changed a great deal (for instance, if its algorithm was optimized).

To merge two or more results, simply multi-select their top-level node in the Explorer panel (use Ctrl or Shift keys for multi-selection) and then choose **Merge** from the Explorer toolbar or from the context menu. You may include records from the Merged Results section in a later merge, since AQtime considers merged results just like other results.

Note that the Merge item of the context menu is enabled only when you select the top-level node of a result set. If you select a category or a thread node, the item will be disabled.

AQtime merges column values according to the column meaning. We will illustrate this using an example. Suppose, you profiled the `fooA` function two times with the Coverage profiler and then merged the results of these two runs. The resultant Hit Count value of the `fooA` function will be the sum of Hit Count values in these two results. If you profiled `fooA` at line level, the lines' Hit Count results will also be summarized. The routine's Lines Covered and Lines Uncovered results will not be summarized. They will be recalculated according to the line's Hit Count values in the merged result. The % Covered result of a routine will be recalculated according to these Lines Covered and Lines Uncovered results. Therefore, it will show what portion of the `fooA` function has been executed after that two runs of the Coverage profiler.

Results of the Performance profiler include columns that indicate the minimum and maximum results for a function, for example, Min Time and Max Time. When merging these columns, AQtime will include the minimum of Min Time and maximum of Max Time values into the merged result set. Columns holding such results as Time or Hit Count will be summarized. Columns holding average values, for example, Average Time, will be recalculated according to the summarized Time and Hit Count values. Percent columns (for example, % Time) will also be recalculated according to the summarized values.

Note: If you merge results of the Performance profiler, be sure they were generated by the same counter. AQtime cannot merge results that were generated by different counters.

Some AQtime profilers, for instance, Performance, Coverage or Function Trace, organize results by threads. The result sets produced by these profilers contain the **All threads** item and the items for each profiled thread. When you merge results of these profilers, AQtime merges the data of the **All threads** items and copies the data of individual thread items. This happens because it's impossible to determine which thread of one set of results corresponds to a thread of another set of results. So, by default, the individaul thread items are not merged. They are just copied to the merged result set.

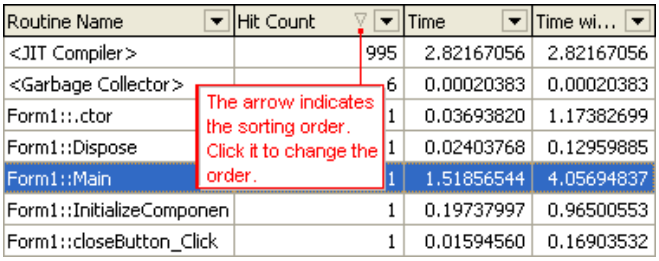
To work around this limitation, you can assign a custom name to the threads in your application. In this case, to identify the thread AQtime will use this custom name rather than the thread id. It will be possible to determine the correspondence between threads in different result sets and AQtime is able to merge them. In other words, AQtime can merge named threads. For information on how to specify custom names for threads, see *Assigning Names to Threads*.

Similarly, to work with ordinary results, you can use both the Summary and Report panels. The Summary panel displays brief profiling results for all merged result sets. For merged results the panel displays session information, profiler-dependent data and profiler options; it does not show run settings and system information sections that are available for ordinary result sets. For the profilers that organize results by threads, the panel shows summary results for all threads and for each individual thread. The Report panel displays detailed information about every profiled routine, object, class and so on. The panel contents is the same as for ordinary result sets, the difference is that individual values are calculated as described above: the actual measurements are summarized, derived characteristics are recalculated, minimum and maximum values are chosen via comparison and so on. Additional information on merged results can also be displayed in the Editor, Call Graph, Call Tree, Details and Disassembler panels.

The Explorer panel includes the *Auto-merge* option group which you can use to merge results automatically. The group includes two options: *Active* and *Folder name*. *Active* lets you turn the auto-merging on or off. *Folder name* specifies the name of a result set in the Merged Results folder with which every new result set with automatically be merged. That is, the Folder name result serves as a cumulative storage to which every new result is added.

Sorting Results

In any panel, on any column where sorting makes sense, you can sort records on that column by clicking the header, once for sorting in one direction, once more to switch directions. The fact that records are sorted on that column, as well as the sort direction, is shown by an arrowhead next to the column caption.



You can sort on several columns in succession. Hold down the Shift key. The first column you click will be the first sort key, the second will be the second key, etc. Re-click on any column without Shift, and it becomes the single sort key again.

The context menu of some AQtime panels offers **Sort Ascending** and **Sort Descending**. This is another way of doing what a simple click or two would do.

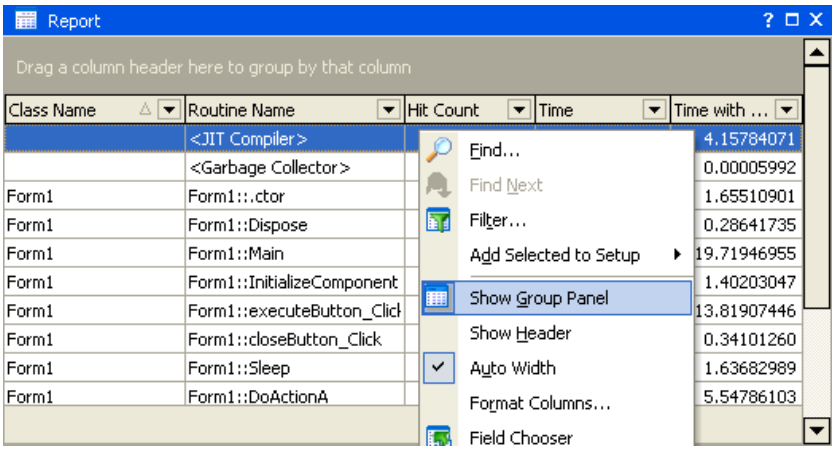
To cancel sorting on a column, press the Ctrl key and then click the header of the appropriate column.

Grouping Results

Grouping results means getting all results (records) that share a single value for one field (for example, **Class**), to show on a single line in the Report panel. The column you choose to group results on becomes a synopsis of the entire result set (for instance, results grouped by class), shown in tree fashion, and the individual records are available by opening up the appropriate branch. This simplifies onscreen navigation of the Report panel when there are several records to show.

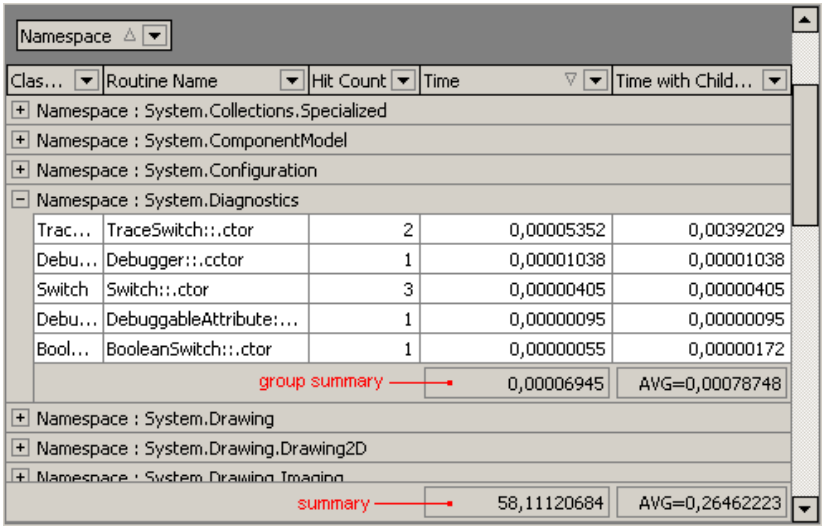
This is one reason all of the profilers include fields (for example, **Source file**, **Class name**, etc.) that help locate a profiled function in source code. For instance, when you group results on the Source file column, each grouped tree node corresponds to a separate source file. Try it!

You can apply grouping simply by choosing **Group by This Column** from the context menu for the column header. However, for better control, and especially to undo grouping, you should choose **Show Group Panel** from the Report toolbar or from the context menu anywhere in the panel. This opens the grouping area:



To group results on a column, drag the column header to the grouping area. To ungroup results, drag the column header out of this area.

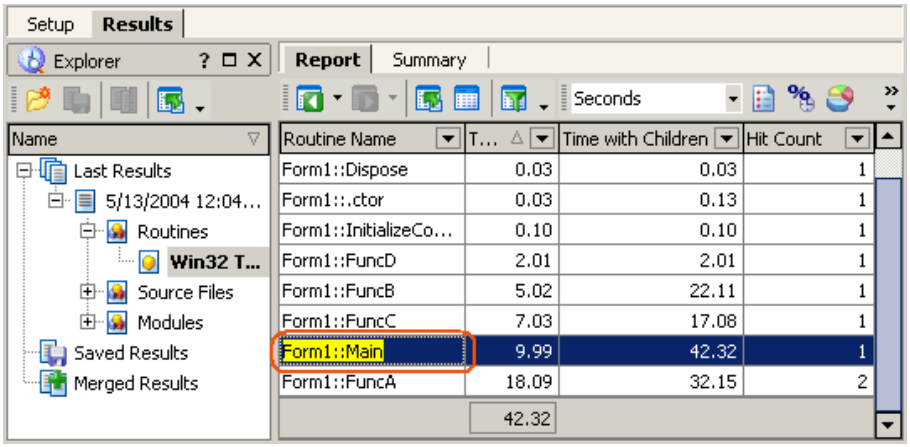
You can group on more than one column. Note that in addition to the "global" summary that is shown at the bottom of the Report panel, AQtime displays "group" summary at the bottom of each group:



Searching Results

AQtime offers two means of searching through the records in a panel.

- You can click on a column header and begin to type the word to search for. Incrementally, the highlight will move to the corresponding record as you add letters. This is case-insensitive.



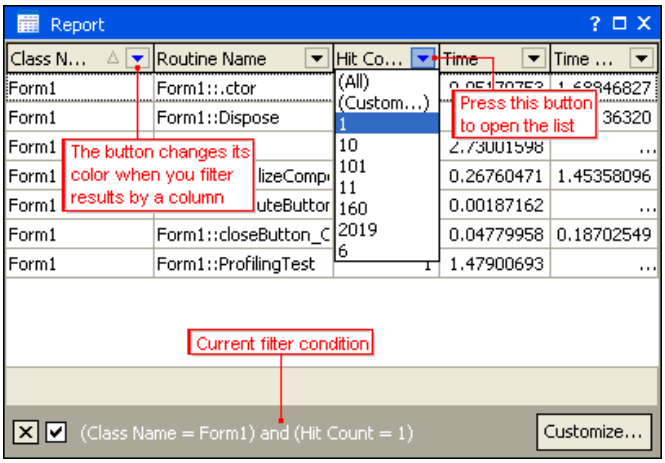
The incremental search also works in panels that do not hold columns. For example, it works in the **Modules** list of the Setup panel.

- In the Report, Event View or Setup panel, choosing **Find** from the context menu or from the Edit menu will bring up the ensuing Find dialog, where you can specify the desired search parameters.

Filtering Results

The output of AQtime's profilers can be displayed in a pre-selected form by defining filters. A filter defines conditions that records (report lines) must meet in order to be displayed in the Report panel.

The easiest way to filter results is to press the dropdown button in the header of a column in the Report panel. Pressing the button opens a list that holds all values displayed in this column. Select the desired value from this list and the Report panel will only display those records which hold the selected value in the column (see the image below). This is also called *auto-filtering*. This is the easiest and fastest way to display results for all of the routines a class, unit or module contains.



You can filter results on several columns: just select the filtering value for each of these columns. For example, on the picture above the filter is applied on the Class Name and Hit Count columns. The current filter conditions are displayed at the bottom of the Report panel. To disable the filter, simply uncheck it there. To remove the filter, press the ☒ button. To modify the filter conditions for the chosen column, select *(Custom...)* from the dropdown list and use the ensuing Custom Filter or Filter dialog. You can also modify the entire filter (for all columns) by pressing the **Customize** button and using the subsequent Filter dialog.

Another way of filtering results is to select **Filter** on the Report toolbar or context menu. This will call the Filter dialog that lets you create custom filter conditions. You can use wildcards in conditions (the _ and % symbols) to search for the desired values in string columns. For more information, see description of the Filter Dialog in on-line help.

Once you have created the desired filter in the dialog, press **OK** to save and apply the filter. You can also save the current filter to or load it from a file.

One final way of filtering results is to use result views. A *result view* combines the filter expression and column layouts in AQtime panels, for one profiler. To apply a result view, simply select it from the **Result Views** dropdown list on the Standard toolbar or from the **View | Result Views** menu (if you use AQtime integrated into Visual Studio or Borland Developer Studio, you can select a view from the Result Views dialog, which is called upon selecting **Profile | Result Views** from Visual Studio's main menu or upon clicking **Result Views** from Borland Developer Studio's **View.AQtime** toolbar). There are several predefined result views for each profiler type. You can also create your own views. For more information on this, see *Result Views*.

Result Views

A **result view** is a group of settings for displaying results. Creating and using result views (including the many pre-defined ones) is a great way to accelerate, simplify and clarify the analysis of your results. A view can serve not only as a preset format, but as a preset question to which you get an immediate, clear answer by switching to the view. Many of the pre-defined result views are of that nature, and you can define more for your own frequently asked questions.

A result view is defined for one specific profiler, and stored in the profiler's single **.qtview** file, which groups all the views currently defined for it. By default, the stored settings are for:

- filter expression
- column layout in AQtime panels
- column layout in the Editor's grid
- panel layout in the Details panel

Some later plug-in profilers may add to the items that are stored in their result views.

A view does not include settings for the panel layout. For more information on how to load and save the panel layout, see *Docking* in on-line help.

You can add a new result view to store your current filter and panel settings, for the current profiler, simply by pressing the **Result Views** button on the Standard toolbar (by selecting **Profile | Result Views** from Visual Studio's menu or by clicking **Result Views** on Borland Developer Studio's **View.AQtime** toolbar) and using the ensuing **Result Views** dialog.

This dialog displays result views available for the current profiler. To apply a view, simply select it in the Result Views dialog and press **OK**. If you use AQtime standalone, you can also select the desired view from the **Result Views** dropdown list on the Standard toolbar as well as in the **View | Result Views** menu.

Until you add your own views, the dialog will display the views that are included with AQtime, and which are defined for all applicable profilers. The rest of this topic is devoted to capsule explanations of these standard views.

The **Default** view is what AQtime uses when executing AQtime for the first time. If you change a series of parameters for the Report panel and then select the Default view, AQtime automatically restores its standard settings.

Current View is simply the name for whatever settings are currently active. It is not a stored view, but you can make it so by saving it and defining a name using the **Result Views** button, to the left of the list. You will then be able to retrieve it for re-use. Note that it applies only to the current profiler.

The **More than 3% (body only)** and **More than 3% (with children)** views are available for the Performance profiler only. You can use them for profiling results that were obtained for any counter the profiler supports.

- For time counters, **More than 3% (body only)** displays the routines that execute the slowest in their own code, independent of the routines they call (*% <Counter-Dependent-Value>* , e.g. *% Time*, is greater than 3).
- For time counters, **More than 3% (with children)** displays the slowest routines, counting all time spent between entry and exit, including "child" calls (*% with Children* is greater than 3).

The Performance profiler supports one more result view - **Default with '%' columns**. This view is similar to the *Default* view. It shows percent columns (for example, *% Time*, *% with Children*, *% Branches*, *% Misses*, etc.) and hides the corresponding non-percent columns (*Time*, *Time with Children*, *Branches*, *Branches with Children*, etc.)

The Coverage profiler supports the following result views:

- **Routines covered less than 50%** - This view lets you quickly isolate routines in which only 50% or less source lines were executed during the profiling run. The Coverage profiler also supports two more similar views: **Routines covered less than 90%** and **Routine covered less than 100%**.
- **Unexecuted routines only** - If you select this view, AQtime will display only those routines that were not executed during the profiler run.

The Static Analysis profiler provides the following result views:

- **Leaf routines (classes)** view. The result of this view depends on the results category that is selected in the Explorer panel. If the *Routines* category is selected, the view will display those routines that do not call other routines. If the *Classes* category is selected, the view displays classes that do not call methods of other classes. After such routines or classes are found, you can quickly exclude them from profiling tasks: select all rows in the Report panel and then use the **Add Selected to Setup** item of the Report context menu to add the selected routines (classes) to an excluding area (see *Adding Selected Routines or Classes to Profiling Areas*).
- The **Non-leaf routines (classes)** view is similar to **Leaf routines (classes)**, but it selects elements by another attribute. If the *Routines* category is selected, this view will display those routines that call other routines. If the *Classes* category is selected, the view will display classes that call methods of other classes.
- The **By module** view displays profiling results grouped by the *Module Name* column.
- **Native code routines and classes** and **.NET code routines and classes** let you view profiling results of native code or .NET routines and classes only.

Printing Profiler Results

If AQtime is running as a standalone application, profiling results displayed in the Report orCall Graph panels can be printed directly:

1. Begin by checking the Print Preview form, using **Print Preview** from the context menu. This does not just preview how results will print out, but allows you to completely configure the printing: color and font settings, printer properties, paper size, background images, etc. Once you are satisfied with the configuration and preview, you can print directly from the form.
2. If you already know how the report will print and do not wish to change anything, you can use **Print** directly from the Report panel context menu as a shortcut.

If you prefer, you can of course print reports indirectly by first exporting results to an external file (XLS, XML, TXT or HTML) and then printing using the appropriate application (for instance, Microsoft Excel or Internet Explorer). This is convenient because your exact printing source remains on file, fully formatted.

Exporting Results

AQtime includes several ways to export data: you can export data that is shown in individual AQtime panels or you can export the entire result sets.

Exporting Entire Result Sets

To export result sets, use items of the Explorer panel's context menu.

Exporting Data to and Importing Them From Binary Files

Using items of the Explorer panel context menu, you can save profiling results to a binary file with the **.aqr** extension and load them from this file. To save results, right-click the desired result set in the panel and then choose **Save to File** from the context menu. **Load From File** will retrieve results saved to a binary file this way.

Exporting Data to a Database

To export a result set to a database, right-click it in the Explorer pane and choose **Export to Database** from the context menu. In the ensuing dialog, specify the database connection settings and click **OK**. For detailed information, see *Exporting Profilnig Results to Database*.

Note that you can also export results to a database by working with AQtime via COM. See *Exporting Results to Database via COM*.

Exporting Data of Individual Panels

Explorer Panel

Using items of the Explorer panel context menu, you can save results to a binary file with the **.aqr** extension and load them from this file. To save results, right-click the desired result set in the panel and then choose **Save to File** from the context menu. **Load From File** will retrieve results saved to a binary file this way.

Report, Details, Disassembler and Event View Panels

From the Report panel profiling results can be copied to the clipboard or exported to a file in any of the following formats: **Microsoft Excel**, **tab-delimited text**, **HTML** or **XML** (viewable in Internet Explorer 5.0 - 7.0, Firefox 1.5 - 2.0, Opera 9.0 or in any browser based on the Microsoft WebBrowser control).

To save results:

3. Select the rows you want to export (you can use Shift and Ctrl for multi-selection. See *Selecting Several Records in a Panel*).
4. Right-click somewhere in the panel and choose **Save Selection** from the context menu. This will open the dialog where you can specify the file format and the file name. After you did these, close the dialog by pressing **Save**.

To save all results (regardless of the selected rows) use the **Save All** context menu item.

To copy results to the clipboard, select the desired rows and then choose **Copy** from the context menu.

When you save or copy the profiling results from the Report panel, column headers are always included in their current order on the panel. When exporting the results of the Performance profiler to XML, AQtime exports both Report and Details results. For a detailed overview of the XML output produced by AQtime, see *Structure of XML Results*. This may be necessary if you want to process the resultant XML files to meet your specific needs.

If you want to only save the Details data (without Report rows), right-click within the desired grid of the Details panel and select **Save All** or **Save Selection** from the context menu.

This is what concerns the Report panel. Besides export of results in the Report panel, you can also export the contents of the Details, Event View and Disassembler panels to the Excel's XLS (Details), XML (Event View and Disassembler), HTML and tab-delimited text files. To do this, multi-select the desired lines in the corresponding panel and then choose **Save Selection** from the panel's context menu. To save all lines rather than selected only, choose the **Save All** menu item.

Call Tree Panel

To store the results displayed in the Call Tree panel, right-click within the panel to invoke the context menu and then choose Save All. This will bring up the dialog where you can specify the name, path and type of the file. The following file formats can be used: XML, HTML and tab-delimited text.

Structure of XML Results

This topic describes the structure of results exported into XML files. This may be necessary if you want to process these files to meet your specific needs.

When exporting results into an XML file, AQtime creates two files:

- **<FileName>.xml**. This file holds profiling results as well as information on their structure.
- **<FileName>.xsl**. This file defines the rules for displaying information from the .xml file.

The .xml file stores data in a tree-like structure:

- (*level 0*) **<AQtime_Results>** - The section of the topmost level. It holds all other sections.
  - (*level 1*) **<HEADERS>** - This section holds information about the results structure. It holds one or several TABLE_HEADER sections.
    - (*level 2*) **<TABLE_HEADER NAME="*table_name*" ID="*nn*">** - For each AQtime data table (Report, Parents, Children, etc.) there is a section like this one. They hold information about columns in the Report and Details panels. *table_name* specifies the data table. *nn* serves as the table id.
      - (*level 3*) **<TABLE_FIELD TYPE="*type*">*field_name*</TABLE_FIELD>** - These tags hold information about the table fields. *field_name* is the name of the field as it is displayed in AQtime panels. *type* specifies the field type (string, integer, float, etc.)
    - (*level 2*) **</TABLE_HEADER>**
  - (*level 1*) **</HEADERS>**
  - (*level 1*) **<RESULTS>** - This section holds profiling results.
    - (*level 2*) **<DATA TABLE_ID="*nn*">** - This section holds data from the data table with id *nn*. The DATA sections that are subsections of RESULTS, hold results displayed in the Report panel.
      - (*level 3*) **<ROWS>** - This tag holds data from a single row of the table.

- (*level 4*) **<FIELD ID="*mm*">*value*</FIELD>** - This tag holds data stored in the table columns. *value* specifies the column value. Boolean values are stored as 0 (False) and 1 (True). Note that fields go in the same order as in which they were listed in the TABLE_HEADER section.

- (*level 3*) **</ROWS>**
- (*level 3*) **<CHILDREN>** - AQtime's Report panel serves as the main result display. Other AQtime panels display additional results for the routine or class that is selected in the Report panel. The CHILDREN section holds addition results.
  - (*level 4*) **<DATA TABLE_ID="*nn*">** - This section holds data from the table with id *nn*. For example, this can be data displayed in the Parents, Children and Lines tables in AQtime. These DATA sections hold the same tags as the DATA sections of the 2nd level.
  - (*level 4*) **</DATA**>
- (level 3) **</CHILDREN>**

- (*level 2*) **</DATA>**

- (*level 1*) **</RESULTS>**

- (*level 0*) </AQtime_Results>

Exporting Profiling Results to Database

Profiling results are represented in a number of AQtime panels. It is useful for visual perception and analysis. To make the profiling results accessible for external applications and to simplify the result processing automation, AQtime provides an option for exporting the profiling results to a SQL Server database. After export the profiling results displayed in AQtime panels are stored in a number of database tables. You can access the data stored in these tables and process it as your needs dictate.

To export profiling results to a database, perform the following actions:

- Right click the desired result set in the Explorer panel and choose **Export to Database** from the context menu. This will call the Export to Database dialog.
- Specify the connection string used to connect to the desired database in the **Connection** box or click the ellipsis to create it via the **Data Link Properties** dialog and specify the user name and password if needed.
- Click **OK** to start exporting.

If you automate AQtime via COM, you can command AQtime to export results by using special interfaces and methods. For detailed information about this, see *Exporting Results to Database via COM*.

The structure of the created database depends on the profiler whose results are exported. For more information on the database structure see the following topics in AQtime help:

- *Allocation Profiler Results - Database Structure*
- *BDE SQL Profiler Results - Database Structure*
- *Coverage Profiler Results - Database Structure*
- *Function Trace Profiler Results - Database Structure*
- *Light Coverage Profiler Results - Database Structure*
- *Load Library Tracer Results - Database Structure*
- *Performance Profiler Results - Database Structure*
- *Platform Compliance Profiler Results - Database Structure*

- *Reference Count Profiler Results - Database Structure*
- *Resource Profiler Results - Database Structure*
- *Static Analysis Profiler Results - Database Structure*
- *Unused VCL Units Profiler Results - Database Structure*

Working with AQtime via COM

Working with AQtime via COM - Overview

AQtime is a COM server and it can be controlled from other applications like any other COM server. Other AutomatedQA applications, for example, TestComplete, use much of it to integrate AQtime tests and results. This topic provides a step-by-step explanation of how you can control AQtime via COM from a Visual Basic application. The topic includes the following sections:

- Setting up Visual Basic (optional)
- Connecting to AQtime
- Using IntegrationManager to Automate AQtime

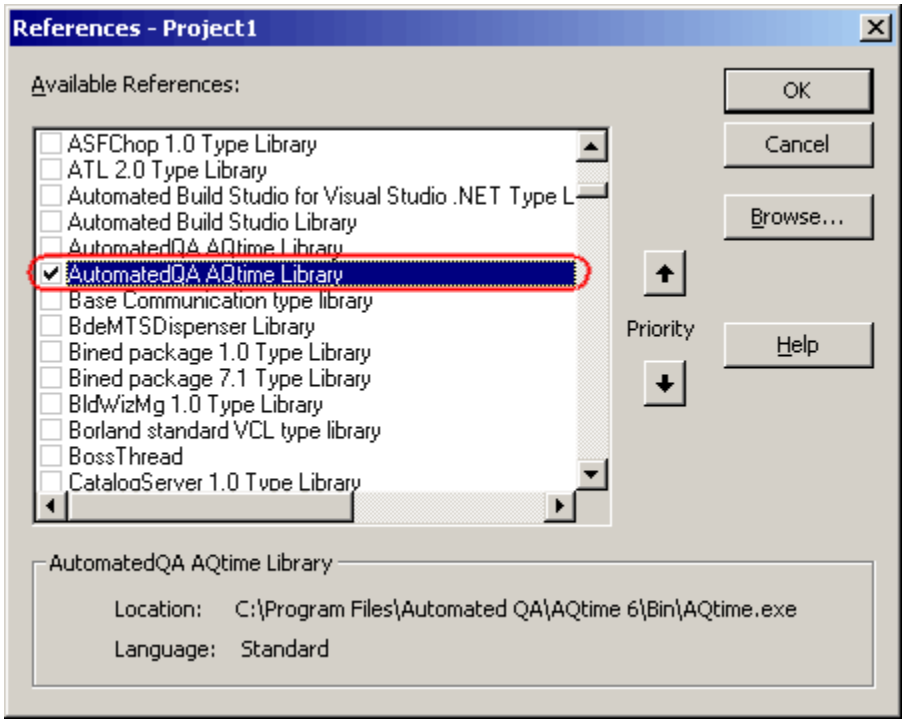
Three notes before we proceed:

- A simple alternative to profiling your applications with AQtime via COM is to run AQtime and pass it certain command-line arguments. See *AQtime Command Line* for more information.
- COM commands work if only one instance of AQtime is running.
- In order to work with AQtime via COM under Windows Vista, Windows Server 2008 or Windows 7, you must work under a user account that has administrator privileges.

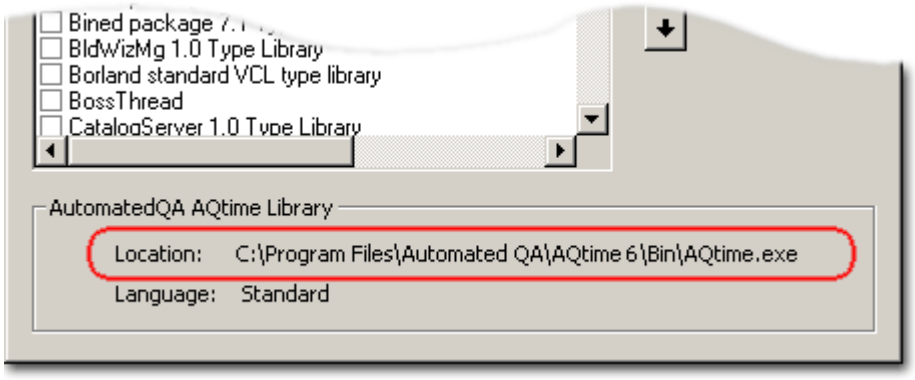
Setting up Visual Basic

To work with AQtime in your VB application, add a reference to the AQtime object library to your application that will use AQtime as a COM server. The following steps explain how you can do this. Note that this step is optional. It is not required if you will work with AQtime via `IntegrationManager` (see below).

- Open your COM client application in Microsoft Visual Basic.
- Select **Project | References** from Visual Basic menu. This will call the **References** dialog.
- In the dialog, check **AutomatedQA AQtime Library** and press **OK**. This will add a reference to the AQtime object library to your COM client application:



Note: AQtime versions 3 and 4 have the same name in References as AQtime 6. To make certain that you selected the latest AQtime object library, have a look at the **Location** field. It should display path to AQtime 6:



If AQtime library is not present in the dialog, you can add it directly by pressing **Browse** and selecting the <AQtime>\Bin\AQtime.exe file in the ensuing Open File dialog.

Connecting to AQtime via COM

The first step in using AQtime as a COM server is connecting to it. To create a connection, you have to create reference to `AQtime` object. This object implements methods and properties to connect and work with AQtime. You can create a connection by calling Visual Basic's `CreateObject` function and passing the ***`AQtime.AQtime`*** or ***AQtime.AQtime.6*** program id to it as a parameter:

```
[Visual Basic]

Set AQtimeObject = CreateObject("AQtime.AQtime")
```

This line will connect your COM client application to the currently running instance of AQtime. If AQtime has not been launched yet, `CreateObject` will launch it.

The `AQtime` object contains the following methods and properties:

Methods and Properties	Description
`Quit`	Method. Use it to exit AQtime.
`Manager`	Property. Returns a reference to the `IaqBaseManager` object that provides access to internal AQtime interfaces. These are the same interfaces that different components of AQtime use to connect and work with each other. Using these interfaces you can perform almost any action that you can do via AQtime's user interface.
`IntegrationManager`	Property. Returns a reference to the `IaqTimeIntegationSupportManager` object (hereinafter we will call it `IntergationManager` for short). It provides access to a subset of higher-level functions specifically engineered to let users perform most typical actions over AQtime with ease.

Functionality provided by `IntegrationManager` is less than what you could do if you would decide to use `Manager`. However, use of `IntegrationManager` is much more easier. For instance, you can start the profiler run and save results by calling only one `IntegrationManager`'s method. If you used `Manager`, you would have to write more complex code.

If you will work with AQtime via the `IntegrationManager` property, you may skip the Setting up Visual Basic step (see above).

Using IntegrationManager to Automate AQtime

The `IntegrationManager` includes methods and properties that let you easily perform typical tasks over AQtime: open a project, select and run a profiler and save profiling results to a file. For detailed information, see *Methods and Properties of the IntegrationManager Object*.

Using the `IntegrationManager` is the easiest way to automate AQtime. Let's create sample code that will:

- Connect to AQtime via COM.
- Open a project.
- Select a profiler.
- Start profiling and save results to an .xml file.

To perform these tasks, we will use only three methods and one property of the `IntegrationManager` object: `OpenProject`, `SelectProfiler`, `Start` and `ProfilingStarted`. The code snippet below also demonstrates how you can specify run parameters:

```
[Visual Basic]

Sub AutomateAQtime()
 Dim AQtimeObject, IntegrationManager, RunMode
 Dim aParamValue, aParamName

 ' Connects to AQtime
 Set AQtimeObject = CreateObject("AQtime.AQtime")

 ' Obtains the IntegrationManager
 Set IntegrationManager = AQtimeObject.IntegrationManager
```

```
' Loads the project
If Not IntegrationManager.OpenProject("C:\Work\MyVCApp\Debug\MyVCApp.aqt") Then
   MsgBox "Cannot open the specified project."
   AQtimeObject.Quit
  Exit Sub
End If

' Selects the desired profiler
If Not IntegrationManager.SelectProfiler("Performance Profiler") Then
   MsgBox "The specified profiler was not found."
   AQtimeObject.Quit
  Exit Sub
End If


' Obtain the object for the "Normal" profiling mode
RunMode = IntegrationManager.GetRunMode(4)

' Get the "Work Directory" parameter
aParamName = ""
aParamValue = Nothing
RunMode.GetParameter(2, aParamName, aParamValue)

' If the parameter is empty assign a value
If aParamValue = Nothing Then
   RunMode.SetParameter(2, "C:\Work")
End If

' Select "Normal" profiling mode
IntegrationManager.SelectRunMode("Normal")

' Add new module to AQtime project
IntegrationManager.AddModule("C:\Work\MyApp.exe")

' Starts profiling and saves results to xml files
Call IntegrationManager.Start("C:\SummaryResults.xml", _
                              "C:\ProfilingResults.xml")

' Waits until profiling is over
While IntegrationManager.ProfilingStarted
   DoEvents
WEnd

' Remove previously added module
IntegrationManager.RemoveModule("C:\Work\MyApp.exe")

' Closes AQtime
 AQtimeObject.Quit
End Sub
```

Two notes:

- Both `Start` and `Attach` methods returns immediately after profiling is started. If you try to close AQtime when it is profiling an application (this happens, for example, when you call `Quit` right after `Start`), AQtime displays a message that asks you whether you want to close application and terminate profiling. To avoid this message, we used a loop that waits until the profiling is over.
- Settings of profiling areas and triggers, selected profiling mode (Normal, ASP.NET, IIS, Service or COM Server), currently selected profiler, and others, are stored in AQtime project files. When you profile your application using the `Start` or `Attach` methods of the `IntegrationManager` object, AQtime uses profiling mode and area and trigger settings stored in your AQtime project.

  Therefore, we recommend to prepare your project in AQtime before you profile that project with AQtime via COM.

© 2010 AutomatedQA Corp.

Working With AQtime From Managed Code

To work with AQtime from managed code, add a reference to the AQtime library to your project:

- Select **Project | Add Reference** from the main menu of Visual Studio. This will call the **Add Reference** dialog.
- In this dialog:
  - Switch to the **COM** tabbed page.
  - Find *AutomatedQA AQtime Library* (TypeLib Version: 6.0) in the list of available components.
  - Choose this library in the list and click the **Select** button. The library will appear in the **Selected Components** list.
  - Press **OK** to close the dialog and save your changes.

After you added the reference, you will have access to AQtime's interfaces. You can automate AQtime through both `IntegrationManager` and `Manager` objects (see *Working With AQtime via COM - Overview*).

The following code demonstrates using of AQtime's `IntegrationManager` via COM:

```
[C#]

using AQtime;
using System.Runtime.InteropServices;
...

private void TestFunc()
{

    const string AQtimeProgID = "AQtime.AQtime.6";
    object AQtimeObject = null;
    object aParamValue = null;
    string aParamName;


    // Obtains access to AQtime
    try
    {
        AQtimeObject = Marshal.GetActiveObject(AQtimeProgID);
    }
    catch
    {
        try
        {
        AQtimeObject = Activator.CreateInstance(Type.GetTypeFromProgID(AQtimeProgID));
        }
        catch
        {
        }
    }

    if (AQtimeObject == null) return;

    // Obtains IAQtimeManager
    AQtime.IAQtimeManager AQtimeManager = (AQtime.IAQtimeManager)AQtimeObject;
    // Obtains IntegrationManager
    AQtime.IaqTimeIntegrationSupportManager IntegrationManager =
AQtimeManager.IntegrationManager;

    // We have a reference to the integration manager.
    // Now we can use its methods and properties to automate AQtime.

    // Loads the project
    if (!IntegrationManager.OpenProject("C:\\Work\\MyVCApp\\Debug\\MyVCApp.aqt"))
```

```
            System.Windows.Forms.MessageBox.Show("Cannot open the specified project.");
        else
        {
            // Selects the desired profiler
            if (!IntegrationManager.SelectProfiler("Allocation Profiler"))
                System.Windows.Forms.MessageBox.Show("The specified profiler was not found.");
            else
            {

                // Obtains aqTimeIntegrationRunMode object for the "Normal" profiling mode
                AQtime.IaqTimeIntegrationRunMode RunMode = IntegrationManager.GetRunMode(4);

                aParamName = "";
                aParamValue = null;
                // Gets the "Work Directory" parameter
                RunMode.GetParameter(2, out aParamName, out aParamValue);

                // If the parameter is empty assigns a value
                if (aParamValue == null) RunMode.SetParameter(2, "C:\\Work");

                // Selects "Normal" profiling mode
                IntegrationManager.SelectRunMode("Normal");

                // Adds new module to AQtime project
                IntegrationManager.AddModule("C:\\Work\\MyApp.exe");


                // Starts profiling and saves results to xml files
                IntegrationManager.Start("C:\\SummaryResults.xml",
"C:\\ProfilingResults.xml");

                // Waits until profiling is over
                while (IntegrationManager.ProfilingStarted)
                    Application.DoEvents();

                // Removes previously added module
                IntegrationManager.RemoveModule("C:\\Work\\MyApp.exe");

            }
        }

        // Closes AQtime
        AQtimeManager.Quit();

        // Releases COM objects
        Marshal.ReleaseComObject(IntegrationManager);
        Marshal.ReleaseComObject(AQtimeManager);
        Marshal.ReleaseComObject(AQtimeObject);
    }
```

Please pay attention to the last three lines. They call the `Marshal.ReleaseCOMObject` method to release all the objects that we referred to in our code. Make sure that you call this method and release the objects. Else, the work with AQtime will not be finished properly and the next attempt to automate AQtime will fail.

Methods and Properties of the IntegrationManager Object

The `IntegrationManager` object is used to automate AQtime via COM. You can obtain a reference to it, by connecting to AQtime's COM engine and calling its `IntegrationManager` property (see *Working With AQtime via COM - Overview*).

The object contains the following methods and properties:

Methods and Properties	Description
`AddModule(`*`AFileName`*`)`	Method. Adds the specified file to the currently opened project.

	The *AFileName* parameter should contain the fully-qualified path to the desired file. If the given file already exists in the project, it is not added. If the file is not found using the specified path, the module is added but marked as unavailable.
`Attach(`*`ProcessId`*`, `*`SummaryResultsFile`*`, `*`FullResultsFile`*`)`	Method. Starts profiling by attaching AQtime to the process with the identifier specified by the ProcessId parameter. The *SummaryResultsFile* and *FullResultsFile* parameters specify the fully qualified names of .xml files, to which results will be stored: • *SummaryResultsFile* is an .xml file holding brief results that are shown in the Summary panel (to convert it to HTML, you can use XSL files from the <AQtime>\Bin\Extensions\Profilers.Files folder). • The *FullResultsFile* file holds all the profiling results. If you do not need any of these two files, simply pass an empty string as a parameter. If the profiling is started successfully, the method returns True; else - False. See also description of the Start method below and the Attach to Process topic. The `Attach` method returns immediately, it does not wait until profiling is over. So, to wait until the end of profiling, check the `ProfilingStarted` property in a loop. Note that attaching to process is available in the **Normal** profiling mode only.
`BaseManager`	Property. Provides program interface to the `IaqBaseManager` object. For more information on using this object, see the *Using IaqBaseManager to automate AQtime* section of this topic.
`ClearResults`	Method. Clears the current result set.
`EnableProfiling(`*`AEnable`*`)`	Method. Turns the profiling status on or off according to the value of the *AEnable* parameter (True or False). If the profiling status is changed successfully, the method returns True; else - False. A calling to this method has the same effect as pressing the **Enable/Disable Profiling** button in AQtime.

`ExportCurrentResults(`*`SummaryResultsFile`*`,`*`FullResultsFile`*`)`	Method. Use this method to export results that are currently shown in AQtime. The SummaryResultsFile and FullResultsFile parameters specify the fully qualified names of .xml files, to which results will be stored: • *SummaryResultsFile* is an .xml file holding brief results that are shown in the Summary panel (to convert it to HTML, you can use XSL files from the <AQtime>\Bin\Extensions\Profilers.Files folder). • The *FullResultsFile* holds all the profiling results. If you do not need any of these two files, simply pass an empty string as a parameter. If the results have been exported successfully, the method returns True, else - False.
`GetRunMode(`*`Index`*`)`	Method. Returns the `aqTimeIntegrationRunMode` object specified by the index. The `aqTimeIntegrationRunMode` object describes the corresponding profiling mode and retrieves and sets the profiling parameters at runtime. The possible indexes are: 0 - Service mode; 1 - COM Server mode; 2 - IIS mode; 3 - ASP.NET mode; 4 - Normal mode.
`NewProjectFromModule(`*`AExecutableFileName`*`)`	Method. Creates a new AQtime project for the specified executable. The *AExecutableFileName* parameter specifies the full name of the desired executable file. If the project is created successfully, the method returns True; else - False. Calling this method has the same effect as selecting **File | New Project From Module** item in AQtime's main menu.
`OpenProject(`*`AProjectFileName`*`)`	Method. Opens an .aqt project in AQtime. The *AProjectFileName* parameter specifies the full name of the desired project file. If the project is opened successfully, the method returns True; else - False.
`ProfilerCategory(`*`Index`*`)`	Property. Returns the category name of a profiler by profiler index in AQtime's profiler collection. Index of the first profiler in the collection is 0. The total number of profilers in the collection is specified by the ProfilersCount property.
`ProfilerName(`*`Index`*`)`	Property. Returns the name of a profiler by profiler index in AQtime's profiler collection. Index of the first profiler in the collection is 0. The total number of profilers in the collection is specified by the ProfilersCount property.
`ProfilersCount`	Property. Returns the total number of available profilers in AQtime.
`ProfilingStarted`	Property. Returns True, if AQtime is running a profiler. Else - False.
`RemoveModule(`*`AFileName`*`)`	Method. Removes the specified file from the currently opened project. The *AFileName* parameter denotes a fully-qualified path to the file to be removed.

© 2010 AutomatedQA Corp.

`ResultReady`	Property. Returns True if AQtime contains result sets to be displayed or exported. Else - False. The property is automatically set to False after the profiling starts and it remains False until the results are generated. The results can be generated during the profiler run by the Get Results menu item or by an action, or at the end of the run by AQtime.
`RunModeCount()`	Method. Returns the number of currently available profiling modes.
`SelectProfiler(`*`AProfilerName`*`)`	Method. Changes the current profiler with a profiler specified by the *AProfilerName* parameter. This parameter should specify the profiler name as it is shown in the **Profilers** box in AQtime. If the profiler is selected successfully, the method returns True; else - False. Note that profiling is started in the currently selected profiling mode. The profiling mode is stored in your AQtime project.
`SelectRunMode(`*`RunModeName`*`)`	Method. Selects the profiling mode with the specified name. The acceptable mode names are: "Service", "COM Server", "IIS", "ASP.NET" and "Normal".
`Start(`*`SummaryResultsFile`*`, `*`FullResultsFile`*`)`	Method. Starts profiling by running the currently selected profiler in AQtime. The *SummaryResultsFile* and *FullResultsFile* parameters specify the fully qualified names of .xml files, to which results will be stored: • *SummaryResultsFile* is an .xml file holding brief results that are shown in the Summary panel (to convert it to HTML, you can use XSL files from the <AQtime>\Bin\Extensions\Profilers.Files folder). • The *FullResultsFile* file holds all the profiling results. If you do not need any of these two files, simply pass an empty string as a parameter. If the profiling is started successfully, the method returns True; else - False. See also description of the `Attach` method. **Note**: the `Start` method returns immediately, it does not wait until profiling is over. So, to wait until the end of profiling, check the `ProfilingStarted` property in a loop. See the example in the *Working With AQtime via COM - Overview* topic.
`TakeSnapshot(`*`Timeout`*`)`	Method. Use this method to generate profiling results. If results are generated successfully, the method returns True; else - False. The *Timeout* parameter specifies the number of milliseconds AQtime waits until the results are generated. If results are not generated within the specified timeout, the method returns False. Calling this method has the same effect as pressing the **Get Results** button in AQtime.

`TerminateProfiling()`	Method. Terminates the application under test without generating profiling results. A call to this method has the same effect as pressing the **Terminate** button in AQtime.

Getting and Settings Profiling Mode Parameters

To obtain program access to profiling mode parameters via COM, connect to AQtime and call the the `GetRunMode` method of the `IntegrationManager` object. This method returns a reference to the special program object, `IaqTimeIntegrationRunMode`, that provides information about the profiling mode and lets you specify profiling parameters. You can find a sample code the demonstrates using of this object in the *Working With AQtime via COM - Overview* topic.

The `IaqTimeIntegrationRunMode` object has the following methods and properties:

Methods and Properties	Description
`GetParameterName(`*`Index`*`)`	Method. Returns the name of a parameter specified by its index. To read a parameter's name when working via COM, you should use this method. The *Index* parameter specifies the desired parameter. *Index* is zero-based and it should be less than the total number of parameters that is returned by the `ParameterCount` property (3 for Normal and COM Server and 0 for ASP.NET, IIS and Service). Otherwise, an error occurs.
`GetParameterValue(`*`Index`*`)`	Method. Returns the value of a parameter specified by its index. To read a parameter's value when working via COM, you should use this method. The *Index* parameter specifies the desired parameter. *Index* is zero-based and it must be less than the total number of parameters that is returned by the `ParameterCount` property (3 for Normal and COM Server and 0 for ASP.NET, IIS and Service). Otherwise, an error occurs.
`Name`	Property. Returns the name of the profiling mode to which the object relates.
`ParametersCount`	Property. Specifies the number of parameters. Each profiling mode, except for "Service", has its own set of run parameters. The Normal and COM Server modes have three parameters, while the ASP.NET and IIS modes have only one parameter. In AQtime, they can be accessed via the Run Parameters dialog.
`SetParameterValue(`*`Index, Value`*`)`	Method. Assigns a new value to the specified parameter. The *Index* parameter defines the parameter to be set. The variant *Value* parameter specifies the value to be assigned.

© 2010 AutomatedQA Corp.

Exporting Results to Database via COM

With AQtime, you can export profiling results to a database. To do this, you call specific commands from the context menu of the Explorer panel and specify the desired database in the ensuing dialog.

You can also use AQtime's COM interfaces to export results to a database. To do this:

- Connect to AQtime via COM.
- Obtain a reference to the `IaqExportResultsToDBManager` object.
- Call the `ExportCurrentResults` method of this object.

The following Visual Basic code demonstrates how you can perform these steps:

```
[Visual Basic]

Sub ExportData
  ' Connect to AQtime via COM
  Set AQtimeObj = CreateObject("AQtime.AQtime")

  ' Open project and start profiling
  Set IntegrationManager = AQtimeObj.IntegrationManager
  IntegrationManager.OpenProject "C:\My_AQtime_Project_exe.aqt"
  IntegrationManager.Start "C:\Results\SummaryResults.xml",
"C:\Results\FullResults.xml"

  ' Wait until profiling is over
  While IntegrationManager.ProfilingStarted
    DoEvents
  WEnd

  ' Get the IaqExportResultsToDBManager object
  Set Mngr =
AQtimeObj.Manager.ManagersByString("{35A743D2-CEA8-4491-A7AA-E1D61468FCE6}")

  ' Export results to a database
  Call Mngr.ExportCurrentResults("Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Export.mdb;Persist Security Info=False", "", "")
End Sub
```

Notes:

- To obtain the `IaqExportResultsToDBManager` object, we use the following code:

  ```
  Set AQtimeObj = CreateObject("AQtime.AQtime")
  ...
  Set Mngr = AQtimeObj.Manager.ManagersByString( _
              "{35A743D2-CEA8-4491-A7AA-E1D61468FCE6}")
  ```

  That is, we obtain this object through the `Manager` property of the AQtime COM object. The `ManagersByString` property returns the desired object by its interface's GUID.

- The `ExportCurrentResults` method has the following syntax:

```
HRESULT ExportCurrentResults(
              [in] BSTR ConnectionString,
              [in] BSTR Login,
              [in] BSTR Password);
```

The *ConnectionString* parameter specifies the string that is used to connect to the database. The content of this string depends on the database type and location. In our example, we export data to a Microsoft Access database (.mdb file). The file name is part of the connection string:

```
Call Mngr.ExportCurrentResults("Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Export.mdb;Persist Security Info=False", "", "")
```

Note that the database (in our case, the Export.mdb file) must exist. The method does not create the database. It exports data into existing databases.

The *Login* and *Password* parameters specify the user account and password that will be used to connect to the database.

Miscellaneous

Integration With Microsoft Visual Studio Team System

Microsoft Visual Studio 2005 Team System, part of Microsoft Visual Studio 2005 and 2008, includes tools and technologies for all members of software development teams: project architects, developers, managers and, of course, testers. AQtime integrates with and supports these tools giving testers and Quality Assurance personnel the possibility:

- To configure AQtime projects and run them directly from Visual Studio's user interface as you would in AQtime when it is running as a standalone application. For this purpose, AQtime panels and toolbars are incorporated into Visual Studio and the main menu of Visual Studio is extended with AQtime-specific items. See *User Interface Overview*.
- To extend your Visual Studio test projects with AQtime projects.

The latter means that your tests created in Visual Studio can run AQtime in order to check the platform compliance of the application under test or to search for memory or resource leaks in the application., which makes your testing flexible and powerful. More details about this are provided below. You can also find information on how AutomatedQA products support Microsoft Visual Studio 2005 at http://www.automatedqa.com/products/team-system-support/.

AQtime includes a special package that integrates into Microsoft Visual Studio 2005 and 2008. After the package is installed, it creates a new test type named **AQtime**. You add test items of this type to your test project created in Visual Studio and run AQtime projects as part of your Visual Studio tests. Here is a step-by-step description of how you can do this.

Note: The integration package can be installed during AQtime's installation. You can select it when AQtime asks you about the plug-ins to be installed. Note that to install the package, Microsoft Visual Studio 2005 Release Candidate (ver. 8.0.50727.26) or later must be installed on your computer.

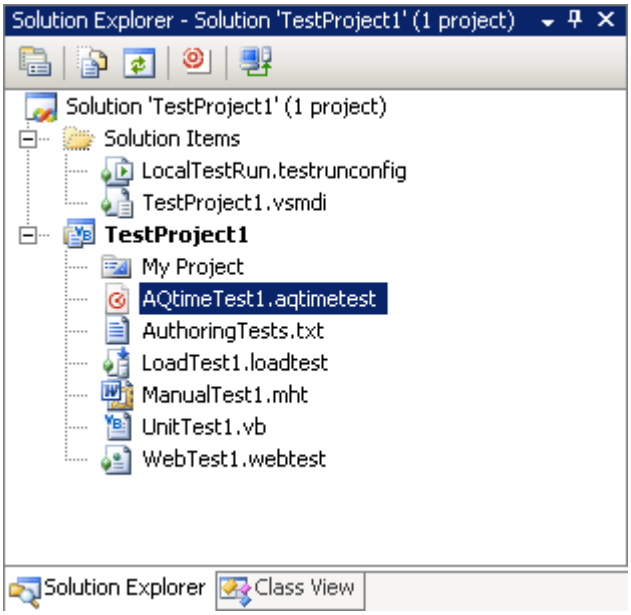
To integrate the package into Visual Studio 2008, Microsoft Visual Studio 2008 ver. 9.0.21022.8 or later must be installed on your computer.

To execute an AQtime project from a test project created in Visual Studio:

1. Prepare your AQtime project for running (specify the modules to be profiled, profiling areas, triggers and actions, and so forth).

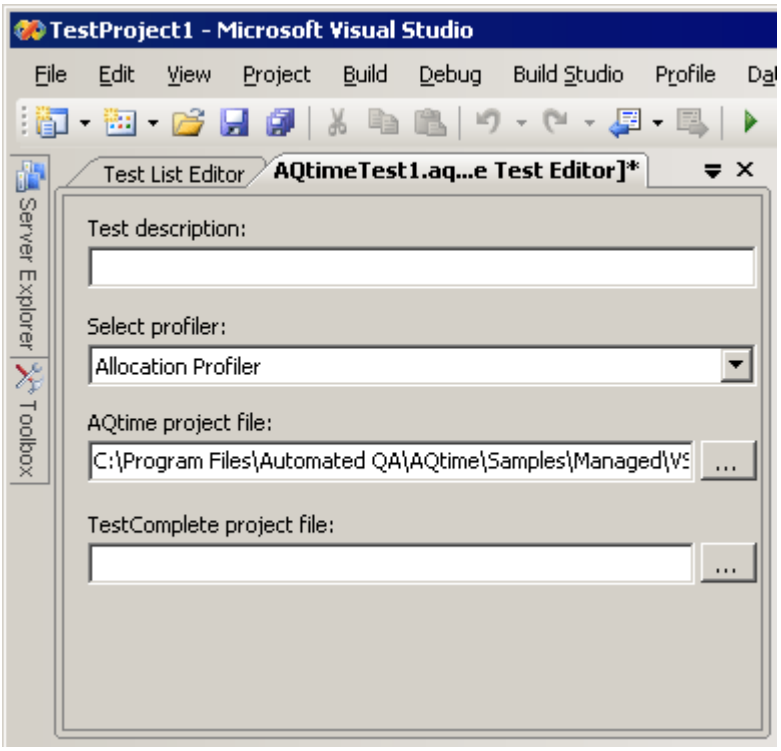
2. Open your Visual Studio test project in Microsoft Visual Studio 2005 (or Visual Studio 2008).
3. Right-click the project node in the **Solution Explorer** and choose **Add | New Test** from the context menu. Visual Studio will display the **Add New Test** dialog displaying the available test types.
4. In the dialog:
   - Select *AQtime Test* in the **Templates** box.
   - Specify the desired test name and the Visual Studio project where the test will be added.
   - Click **OK**.

   Visual Studio will create the new test of the *AQtime* type and will show it as a child node of the project's node in the Solution Explorer.



Now we can modify properties of our test. To do this:

5. Double-click the *AQtime* test node in the Solution Explorer. Visual Studio will display the item editor.
6. Switch to the test editor.



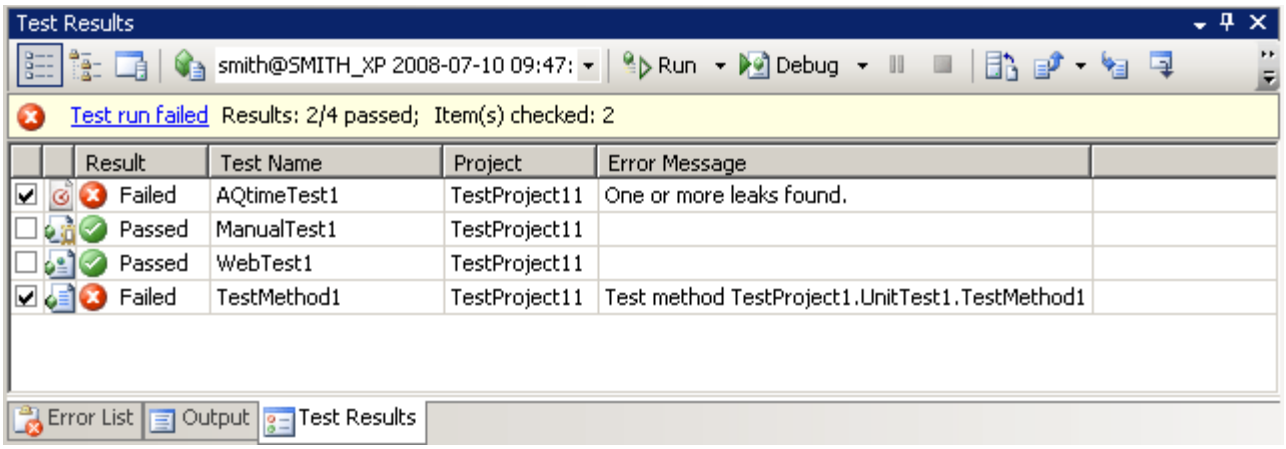
7. (Optional step.) Specify test goals, details or other information regarging the test in the **Test description** edit box.
8. In the **Select profiler** dropdown list box, choose the profiler that will be used for your tests. You can select one of the following profilers: *Allocation Profiler*, *Resource Profiler* or *Platform Compliance*.
9. In the **AQtime project file** edit box, click the ellipsis button and choose the AQtime project file (.aqt) you would like to run.
10. (Optional step.) In the **TestComplete project file** edit box, click the ellipsis button and choose the TestComplete project that will be run to simulate user actions (mouse clicks, keypresses, etc.) over the application under test. When TestComplete is used together with AQtime, user action is not required to perform tests.
11. Select **File | Save All** from Visual Studio's menu to save your changes.

Now you can add the *AQtime* test to the list of tests to be run:

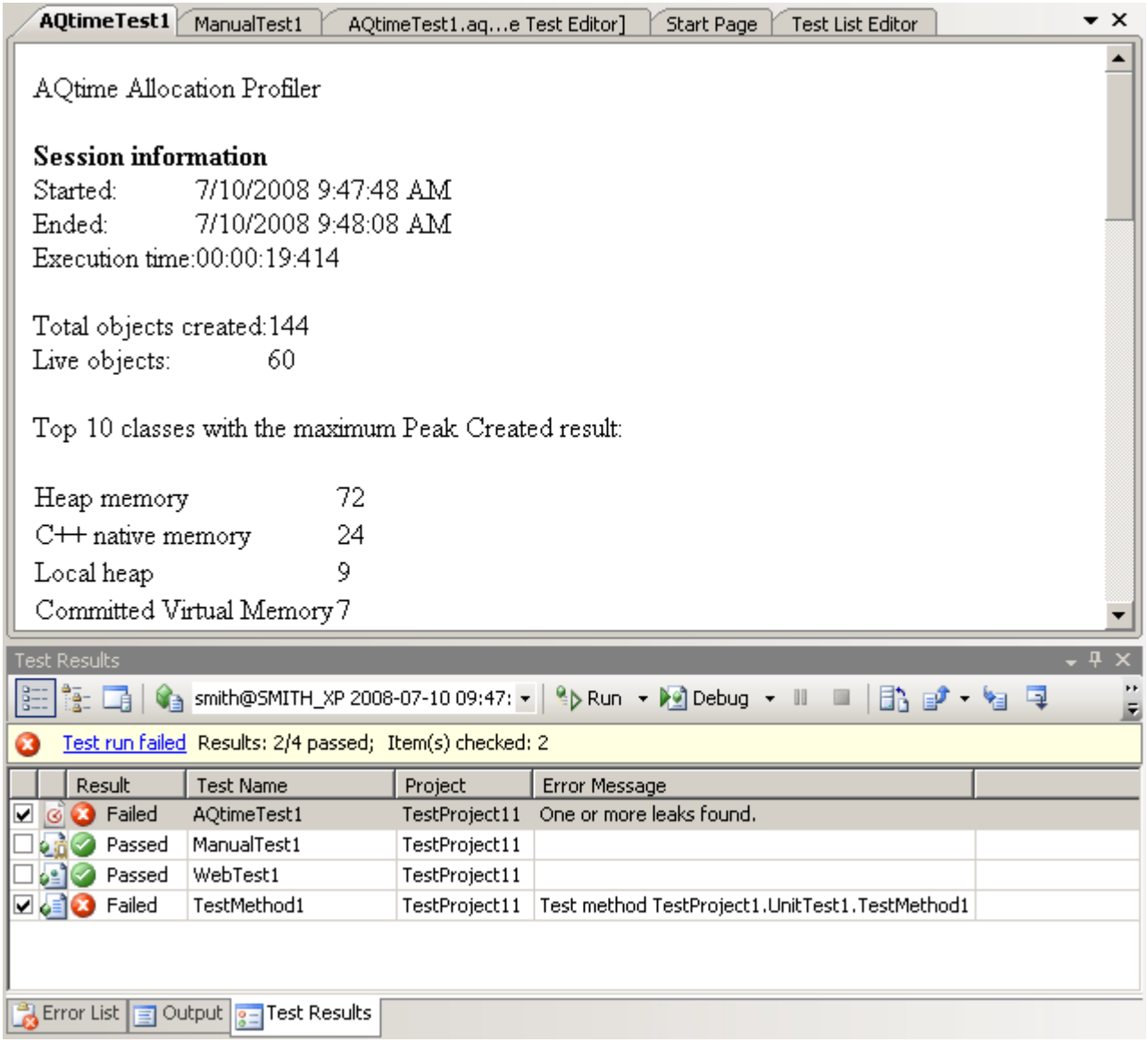
12. If you use Visual Studio 2005, open the **Test Manager** panel (**Tests | Windows | Test Manager**). In Visual Studio 2008 this panel has been renamed to **Test List Editor**. To invoke it, select **Tests | Windows | Test List Editor** from Visual Studio's main menu.
13. Place the *AQtime* test item in the desired category and enable it for execution.

After you have added the AQtime test to the list of tests to be executed, you can run your tests. To do this, click the **Run Checked Tests** button on the Visual Studio toolbar. When the AQtime test item is being executed, it launches AQtime, loads the specified AQtime project, runs the chosen profiler and optionally performs user actions with the application under test via TestComplete.

After the test run is completed, the results of the *AQtime* test item execution are shown in the **Test Results** panel along with any other testing results.



Here you can quickly see whether the AQtime tests passed successfully or failed. To view a detailed output of an AQtime test, simply double-click it in the panel.



AQtime Command Line

AQtime uses the following command-line arguments:

```
<AQtime.exe> (project_name | ( file_name [/s | /SearchProject]
) )
[ ([/r | /run] | [/a:PID | /attach:PID]) [/e|/exit] ]
[ (/silentmode | /errorandwarningmode |  /erroronlymode)]
```

Here the parenthesis means the group of command-line arguments. The "|" means OR, for example, "`/r | /run`" means you can use either `/r` OR `/run`. Square brackets mean the argument or group of arguments is optional. Below is the detailed description of AQtime's command-line arguments:

Argument	Description
project_name	Loads the specified project (.aqt) in AQtime. Do not forget to enclose *project_name* in quotes in case the project name or path contains spaces.
file_name	If `/s` or `/SearchProject` is specified, AQtime searches for the project that matches the specified file name and if the project is found, loads it. If `/s` or `/SearchProject` is specified, AQtime searches for a project that matches the specified file name and if the project is found, loads it. Note that AQtime searches for the project using the "default" project name and the path specified in *file_name*. For instance, if you specify *C:/Work/MyApp.exe* as *file_name*, AQtime will search for the project *MyApp_exe.aqt* in the *C:\Work* folder. Do not forget to enclose *file_name* in quotes in case the file name or path contains spaces.
`/s, /SearchProject`	`/s` and `/SearchProject` are equivalent. They specify whether AQtime should search for and load the AQtime project that corresponds to the executable specified by the *file_name* command-line argument.
`/r, /run`	`/r` and `/run` are equivalent. They command AQtime to start profiling upon loading the project (file) in AQtime. If you load a project, AQtime will profile with the profiler specified in the project file (this is the profiler you used last for your project). In addition, AQtime will use the area, trigger and action settings stored in the project. If you load a file, AQtime will profile with the Performance profiler and with default area and trigger settings (that is, no triggers and actions, *Full Check by Routines* and *Profile Entire .NET Code by Routines* are enabled).
`/a:PID, /attach:PID`	Commands AQtime to profile using the "Attach to Process" feature. *PID* specifies the ID of the process to which AQtime should attach. You can learn the project ID, for example, from Windows' Task Manager. `/a` and `/attach` are equivalent. They are ignored if `/r` or `/run` is specified.
`/e, /exit`	`/e` and `/exit` are equivalent. They command AQtime to close automatically one the profiling has finished. This argument can be used if either, `/r`, /run, `/a` or `/attach` is used. Otherwise, the argument is ignored.
`/SilentMode, /ErrorAndWarningMode,`	You can use one of these arguments to specify what message boxes and dialogs AQtime will display during its work:

`/ErrorOnlyMode`	• `/SilentMode` - If this argument is specified, AQtime will not display any dialogs. For instance, it will not display the Run Settings dialog upon starting the profiling. In addition, it will not inform you of any errors and warnings that may occur during its work. • `/ErrorAndWarningMode` - If this argument is used, AQtime will not display dialogs, but will show error and warning messages. • `/ErrorOnlyMode` - If this argument is used, AQtime will not display any dialogs and message boxes except for the error messages.

Here are some examples of running AQtime with command-line arguments:

- Runs AQtime, loads the specified project (MyApp.aqt) in it and starts profiling:

  "C:\Work\Automated QA\AQtime\Bin\AQtime.exe" C:\MyApp\MyApp.aqt /r

- Runs AQtime, loads the specified project (MyApp.aqt) and starts profiling the project using the the “Attach to Process” feature:

  AQtime.exe C:\MyApp\MyApp.exe /a:1296

As you can see, using command-line arguments you can launch AQtime, load a project in it, start profiling and close AQtime once the profiling has finished. All these give you the ability to integrate AQtime in automated tests of your application. Running these tests after each application build, for instance, you can easily see if changes to the application’s code caused performance bottlenecks or inefficient use of memory. The current version of AQtime cannot simulate user actions (keypresses, mouse clicks, etc.) after profiling starts. However, you can easily do this with TestComplete - a tool that was specially designed for testing Windows applications. For more information on it, see *TestComplete Information*. The use of AQtime along with TestComplete lets you significantly reduce the amount of time spent for testing and managing the application delivery process.

Working With Source Control Systems

Microsoft Visual Studio and Embarcadero RAD Studio (as well as earlier versions of RAD Studio by CodeGear and Borland) provide built-in support for source control systems. This means you can check your files from and check them in a source control directly from your IDE. So, if you use AQtime integrated into Visual Studio or RAD Studio, you can check in and check out AQtime projects in the same manner as you would with other source files.

AQtime standalone also provides integrated source control services. This means that you can use AQtime as a client for your source control provider. Without leaving AQtime, you can check in and check out your AQtime project, view its history, compare its versions and so on. To perform these actions, AQtime, that is running as a standalone application, includes the **File | Source Control** menu.

When AQtime is running as a standalone application, it supports all Microsoft Source Code Control API compatible source control systems, including:

- Microsoft Visual SourceSafe
- Concurrent Versions System (CVS), through the CVSProxy or Jalindi Igloo plug-in
- Merant PVCS Version Manager
- Team Coherence

- Rational ClearCase
- Starbase Versions (StarTeam)
- Seapine Surround SCM
- IBM VisualAge TeamConnection
- CS-RCS
- MKS Source Integrity
- Perforce P4 Version Control System
- CM Synergy (Continuus CM)
- AllChange
- Code Co-op
- SourceOffSite
- AllFusion Harvest Change Manager
- Etc.

Before you start using a source control system, you should add your AQtime project to it. To do this:

- Save your current AQtime project to a file if you have not saved it yet.
- Select **Add to Source Control** from the **File** | **Source Control** menu. This will call the **Select SCC Provider** dialog that lists source control systems installed on your computer. AQtime generates this list automatically.
- Select the desired source control system in the dialog and press **OK**.

Once you press OK in the Select SCC Provider dialog, AQtime will connect to the selected source control system. Then, it will show the source control system's dialog in which you need to specify the project to which you would like to add your AQtime project.

Note: If your AQtime project has already been added to the chosen source control system outside of AQtime, but you want to check it in or out from AQtime, you need to add your project to this source control system anyway as it is described above.

After your AQtime project is added to the source control system, you will be able to check it in or out, view history and use other services provided by the source control system. These actions will be applied to the selected source control system only. If you want to add your AQtime project to another source control system, you must unbind the project from the chosen source control system first. To do this, select **File | Source Control | Unbind From Source Control**. Then you can add your project to another source control system as it is described above.

The File | Source Control menu holds some more items to work with source control systems:

- **Check In** - Checks in your AQtime project to the source control system chosen in the Select SCC Provider dialog.
- **Check Out** - Checks out your project from the selected source control system.
- **Undo Check Out** - Lets you undo the check out. This item is enabled only when Check Out is disabled.

  Note: AQtime traces the check-in/check-out status of an AQtime project and enables or disables the Check In and Check Out methods according to this status. For instance, AQtime disables the Check In item after the project was checked in and enables it after the project was checked out. The Check Out item is disabled when Check In is enabled

and vice versa. However, AQtime's check-in/check-out status of an AQtime project may not coincide with the status of the project in the source control system. This happens if you checked your project out from the source control system using AQtime and then checked it in outside of AQtime using some other tool, for example, using a client application of the source control system. To bring AQtime's status of the project to the source control system's status, select **Refresh Status** from the File | Source Control menu.

- **History** - Calls the History dialog provided by the source control system. This dialog displays the history of the selected file (creation date, check in dates, etc.) and the current version number.
- **Compare Files** - Lets you compare a file stored in the source control system with another file stored in the source control system, or a file stored in the source control system with a file that is not stored in it, or two files that are not stored in the source control system. The actual actions performed by this menu item depend on the source control system you use as well as on the "nature" of the files.
- **Properties** - Invokes the dialog that shows source-control properties of your AQtime project. This dialog is provided by the source control system.
- **Run Application** - Launches the client application of the source control system selected in the Select SCC Provider dialog. This can be necessary if you want to do something that cannot be done with AQtime.

For more information, see documentation on the client that supports the Source Code Control technology.

Visual Studio to which AQtime can be integrated also has the **File | Source Control** menu with similar items. However, their purpose differs from what is described above. In particular, only integration with Microsoft Visual SourceSafe is supported there. To learn more about Visual Studio's native support for source control systems, see the Visual Studio documentation.

Similarly, RAD Studio, into which AQtime can be integrated, supports integration with the Borland StarTeam source control. To learn more about this, see the RAD Studio documentation.

TestComplete Information

TestComplete is AutomatedQA's test management and automation tool for Windows applications. It is tightly integrated with AQtime. TestComplete provides a huge set of features:

- Automated functional, unit, regression, data-driven and distributed testing of Windows applications and Web pages. Performance, load, stress and scalability testing of Web servers.
- Recording and manual creation of testing scripts in VBScript, JScript, DelphiScript or C++Script.
- Access to application object properties, fields and methods directly from script routines. This applies to applications created with any .NET compiler (Microsoft C#, VB.NET, Borland C#Builder, etc.) as well as to applications created with non-.NET compilers (Microsoft Visual C++, Visual Basic, Visual J++, Borland Delphi, C++Builder and JBuilder). TestComplete scripts can set properties and call methods of objects existing in the tested C# application. There is no need to recompile your .NET application for TestComplete to have access to internal application objects, methods and properties.
- The capacity to import test scripts into source code, or write them directly in the program source, and run them from there.
- Debugging of script procedures from directly within TestComplete.
- Handling events of ActiveX objects in scripts.

- A store for files, pictures, object fields and properties necessary for testing. Therefore, while running a test you can compare files, images and properties with the ones you have stored.
- Exporting of test results to XML, XSL or HTML files, in addition to full access, filtering and viewing in TestComplete's Test Log.
- Integration with AQtime. You can run the AQtime profilers from TestComplete; the application will be profiled while it is being run by TestComplete and its results reported to its Test Log.
- An open COM-based architecture. You can create your own plug-ins for special needs. TestComplete includes two such plug-ins for ADO and BDE monitoring, with available source code. The plug-ins allow tests to check on database status from outside the application under test.

For more information on TestComplete, see http://www.automatedqa.com/products/testcomplete/.

AQtrace Information

AQtrace is AutomatedQA's error reporting and resolution system that provides an easy way for software developers to find the cause of errors that occur in their applications running on end-user computers.

AQtrace includes a special component that is integrated into your application and monitors the application execution. When an exception occurs, the component automatically generates an error report, displays a notification window on screen and sends the report and user comments to your server. On the server side, the report is received and saved to the storage for further analysis. The developers can then retrieve the report, investigate the problem and fix it.

AQtrace provides a vast amount of features:

- AQtrace includes special features for tracing application execution, generating error reports, sending theses reports to developers, receiving the reports and analyzing them. It saves the time and energy needed to add the report generating functionality to your applications and to implement the desired reporting and error processing policies.
- AQtrace is a well-organized automated approach to reporting and fixing bugs. It helps developers resolve customers' problems faster and improve the quality of your products.
- AQtrace includes special plug-ins that automatically store the received error reports to issue-tracking systems like AQdevTeam, Visual Studio Team System or Bugzilla.
- AQtrace automatically detects duplicated error reports, so you do not have to analyze and fix the same error over and over again.
- AQtrace can be used with Visual C++, Visual Basic, Delphi, C++Builder and Intel C++ applications. It also supports 64-bit applications.
- AQtrace can catch any exception that occurs in your application regardless of whether the exception occurs within or outside of the `try... catch` block.
- You can define filters and only trace specific kinds of exceptions and specific modules.
- The report contents are customizable. AQtrace offers a rich set of settings that specify information to be included into the generated reports: information about the operating system, service packs and .NET Framework versions, information about the processes, hard-disk space and memory, and so on. You can also include custom information like the internal state of your applications' subsystems or internal applications' flags.

For more information on AQtrace, see http://www.automatedqa.com/products/aqtrace/.

AQtime Data Files

AQtime projects are stored as **.aqt** files. These files include the following information:

- The list of profiled modules
- Profiling results
- Settings of profiling areas, triggers and actions
- The selected profiling mode and the selected profiler
- The list of files to be excluded from profiling (for project) and project search directories

AQtime saves these settings to a project file when you close the AQtime project and loads them from it when you open your project.

The following files are used only if AQtime runs as a standalone application:

- In the <AQtime>\Views folder, there are *profiler_name***.qtview** files. They store result views defined for each profiler. See Result Views.
- The current panel layout on the desktop can be saved anywhere you wish as a **.qtdock** file. To save the panel layout, use **View | Desktop | Save Docking to File** from AQtime's main menu. To load the layout from a file, use **View | Desktop | Load Docking from File**.
- The current toolbar settings can be saved to an **.aqtlb** file by selecting **View | Desktop | Save Toolbar to File** from AQtime's main menu. To load the toolbar settings from a file, use **View | Desktop | Load Toolbar from File**.

Index

.aqr files 497
.aqt files 120, 520
.aqtlb files 520
.NET applications -- Peculiarities of profiling 486
.NET Framework assemblies -- Profiling of 464
.NET runtime --Profiling of 240
.qtdock files 520
<Garbage collector> routine in profiler results 213
<JIT compiler> routine in profiler results 213
<Root> routine in profiling results 215
5 bytes restriction 470
64-bit applications -- Profiling of 404
64K Aliasing Conflicts counter 383

A

Actions 135
Activate after loading -- option 201
Activate on generating results -- option 196
Active -- option 150, 176, 183, 187
Active counter -- option 239, 299
Add to Area -- menu item 486
Adding 487, 497
  Results to source files 497
  Routines and classes to the Setup panel 487
AddModule method 509
Aliasing Conflicts counter 383
All Memory Page Faults counter 384
Allocation profiler 242
  About Allocation profiler 242
  Analyzing Allocation profiler results 256
  Analyzing Borland C++ applications 252
  Analyzing C++Builder applications 250
  Analyzing Delphi applications 248
  Analyzing GNU CC applications 252
  Analyzing Intel C++ applications 252
  Analyzing Visual Basic Applications 247
  Analyzing Visual C++ Applications 245
  Checking bounds of memory blocks 254
  Columns of the Details and Call Tree panels 267
  Options 268
  Report panel columns 265
  Tracing attempts to access released memory 253
  Tracing system management memory functions 253
  Using with the Monitor panel 187
Always set up Compare parameters -- option 183
Analyzing profiler results 141
AQCommunity 23
aqIPD.sys driver
  Problems with Driver Verifier 385
aqr files 497
aqt files 120, 520
AQtime 10
  AQtime Panels 33
  AQtime Profilers 38
  Command line 515
  Community site 23
  Data files 520
  FAQ 23
  Getting started 25
  Getting support 23
  Newsgroups 23
  Overview -- General 10
  System requirements 13
  Using with Windows DDK 15
  What's new 20
  Working via COM 500
  x86 and x64 AQtime Packages 13
AQtime object 501
AQtime panels 33
AQtime profilers 38
  Allocation profiler 242
  BDE SQL profiler 270
  Coverage profiler 274
  Exception Trace profiler 286
  Function Trace profiler 286
  Light Coverage profiler 300
  Load LibraryTracer 309
  Performance profiler 203
  Platform Compliance profiler 322
  Reference Count profiler 316
  Resource profiler 326
  Sequence Diagram Link profiler 347
  Static Analysis profiler 350
  Unused VCL Units profiler 376
aqtlb files 520
AQtrace information 519
Areas 124, 127
  Checking elements to profile 131
  Description 127
ASP.NET applications 45, 408
  ASP.NET or IIS is running under a user account. 430
  Preparing ASP.NET applications 45
  Profiling of 408
  Profiling via ASP.NET Development Server 410
  Profiling via Internet Information Services 413

ASP.NET Development Server 410
Assingning names to threads 483
Assistant Panel 146
Description 146
Options 146
Attach method 505
Attaching to process 402
Auto expand -- option 160
Auto-expand critical path to depth -- option 156
Auto-expand depth -- option 156
Auto-merge -- option 183
Auto-select new elements -- option 201

B

Background color -- option 161
BaseManager property 505
BDE SQL profiler 270
About BDE SQL profiler 270
Description of results 270
Details panel columns 273
Options 273
Report panel columns 272
Blocked Store Forwards Replays counter 383
Borland C#Builder 64
Preparing C#Builder 2006 applications 64
Preparing C#Builder applications 64
Borland C++ 116
Analyzing Borland C++ applications with the Allocation profiler 252
Preparing Borland C++ applications 116
Borland C++Builder
Analyzing C++Builder applications with the Allocation profiler 250
Borland Delphi
Analyzing Delphi applications with the Allocation profiler 248
Preparing Borland Delphi 2005 applications 66
Preparing Delphi 2005 for Win32 applications 84
Preparing Delphi 2006 for .NET applications 65
Preparing Delphi 2006 for Win32 applications 81
Preparing Delphi 3-7 applications 84
Preparing Delphi for .NET applications 67
Borland Developer Studio
Integration into 16
Bounds of memory blocks – Checking 254

C

C# 45, 48
Preparing C# .NET applications 48
Preparing C# 2005 and 2008 applications 45
Preparing C#Builder 2006 applications 64
Preparing C#Builder applications 64
Supported versions 15
C#Builder 64
Preparing C#Builder 2006 applications 64
Preparing C#Builder applications 64
Supported versions 15
C++ 58, 61, 68, 86, 93, 98, 106, 113, 116, 117
Analyzing Visual C++ applications with the Allocation profiler 245
Analyzing C++Builder applications with the Allocation profiler 250, 252
Analyzing GNU CC applications with the Allocation profiler 252
Analyzing Intel C++ applications with the Allocation profiler 252
Preapring Intel C++ applications 117
Preparing Borland C++ applications 116
Preparing C++Builder 2006 applications 106
Preparing C++Builder 2007 applications 98
Preparing C++Builder 2009 applications 93
Preparing C++Builder 2010 applications 86
Preparing C++Builder 3-6 applications 113
Preparing GNU CC applications 117
Preparing Visual C++ .NET applications 61
Preparing Visual C++ 2005 and 2008 applications 58
Preparing Visual C++ 6 (or earlier) applications 68
Supported compilers 15
C++Builder
Analyzing C++Builder applications with the Allocation profiler 250
Preparing C++Builder 2006 applications 106
Preparing C++Builder 2007 applications 98
Preparing C++Builder 2009 applications 93
Preparing C++Builder 2010 applications 86
Preparing C++Builder 3-6 applications 113
Supported versions 15
Calculate % with children relative to real values -- option and toolbar item 240
Calculating percent in the Report panel 212
Call Graph Panel 147
Description 147
Options 149
Call stack 179, 256
Allocation profiler results 256
in the Event View panel 179
Call Tree panel 150
Allocation profiler results 267
Description 150
Options 155
Reference Count profiler results 320
Static Analysis profiler results 374
Character set -- Editor setting 168
Check memory bounds -- option 269
Check system memory allocations -- option 269
Checking bounds of memory blocks 254
Child functions 477
Profiling along with parents 477

Child item color -- option 150
Class level 126
Clear on application start -- option 175
Clear Results -- menu item 401
Clearing results during profiling 401
ClearResults method 505
CLR integration assemblies
  Profiling of 451
Code Editor 164
  Add code to profiling from Code Editor 486
CodeGear
  Integration into RAD Studio 16
  Preparing C++Builder 2006 applications 106
  Preparing C++Builder 2007 applications 98
  Preparing C++Builder 2009 applications 93
  Preparing Delphi 2007 applications 78
  Preparing Delphi 2009 applications 75
  Supported compilers 15
CodeSite 1 output -- option 299
CodeSite 2/3 output -- option 299
Collect stack information -- option 269, 337
COM -- Working with AQtime via COM 500
  About working with AQtime via COM 500
  Export results to database via COM 510
  Getting and setting profiling mode parameters 509
  Members of the IntegrationManager class 505
  Working from .NET applications 504
COM applications 431, 432, 433, 434, 435
  About COM applications profiling 431
  In-process COM servers 432
  Out-of-process COM servers 433
  Profiling COM logical threads 438
  Profiling COM+ and MTS applications 435
  Profiling DCOM servers 434
COM Resources -- option value 337
Command-line arguments 515
Community site 23
Compaq Visual Fortran 118
  Preparing Visual Fortran applications 118
  Supported versions 15
Comparing results 488
Compiler settings
  ASP.NET 45
  Borland C++ 116
  C# -- Visual Studio .NET 48
  C# -- Visual Studio 2005 and 2008 45
  C#Builder 64
  C#Builder 2006 64
  C++ -- Visual Studio 2005 and 2008 58
  C++Builder 3-6 113
  C++Builder 2006 106
  C++Builder 2007 98
  C++Builder 2009 93
  C++Builder 2010 86
  Delphi 3-7 84
  Delphi 2005 ..... 66, 84
  Delphi 2006 for .NET 65
  Delphi 2006 for Win32 81
  Delphi 2007 for Win32 78
  Delphi 2009 for Win32 75
  Delphi 2010 for Win32 72
  Delphi for .NET 67
  GNU CC 117
  Intel C++ 117
  J# -- Visual Studio .NET 52
  J# -- Visual Studio 2005 and 2008 49
  Microsoft Visual Basic 71
  Microsoft Visual C++ 6 or earlier 68
  Visual Basic .NET 57
  Visual Basic 2005 and 2008 53
  Visual C# 2005 and 2008 45
  Visual C#.NET 48
  Visual C++ -- Visual Studio .NET 61
  Visual C++ 2005 and 2008 58
  Visual Fortran 118
  Visual J# .NET 52
  Visual J# 2005 and 2008 49
Completely executed -- option value 285
Compliance level -- option 326
Conflicts -- 64K Aliasing Conflicts counter 383
Content settings -- option 161
Context Switches counter 384
Control characters -- Editor setting 168
Controlling profiling from application code 481
Copying results 497
Counter unit -- option and toolbar item 240, 273, 299
Counters
  Descriptions 381
  General Information 381
  Limitations 385
  Problems (error description on blue screen 385
Counters Overview 380
Coverage profiler 274
  About Coverage profiler 274
  Description of the Details panel columns 284
  Description of the Report panel columns 282
  Options 285
CPU Cache Misses counter 382
CPU Mispredicted Branches counter 383
Creating a project 120
critical path 147, 150
Current View 495
Customization capabilities 492
  Grouping 492
  Sorting 492

D

Database -- Exporting profiling results to 499
DCOM applications -- Profiling of 434
DDK
  Using AQtime with Windows DDK 15

Debug Events -- option....175
Debug info files
Specifying path to....485
Debug information
How AQtime profilers use it....43
Specifying path to debug info files....485
Default View....495
Delphi
Analyzing with the Allocation profiler....248
Preparing Delphi 2005 applications....66
Preparing Delphi 2005 for Win32 applications....84
Preparing Delphi 2006 for .NET applications....65
Preparing Delphi 2006 for Win32 applications....81
Preparing Delphi 2007 for Win32 applications....78
Preparing Delphi 2009 for Win32 applications....75
Preparing Delphi 2010 for Win32 applications....72
Preparing Delphi 3-7 applications....84
Preparing Delphi for .NET applications....67
Supported versions....15
Delphi version -- option....380
Depth shown -- option....175, 176
Details panel....156
Allocation profiler results....267
BDE SQL profiler results....273
Coverage profiler results....284
Description....156
Function Trace profiler results....295
Light Coverage profiler results....308
Load Library Tracer results....314
Options....157
Performance profiler results....224
Reference Count profiler results....320
Resource profiler results....335
Static Analysis profiler results....374
Unused VCL Units profiler results....379
Details part -- option....150
Disable inlining -- option....239, 285
Disable profiling from application code....481
Disabling inlining for the managed application to be profiled....388
Disassembler panel....157
Description....157
Options....160
Disassembly lines -- option....161
Display code -- option....161
Display events in -- option....176
Display magnifier -- option....187
Display settings (Editor settings)....168
DLLs
Profiling of....405
Using custom DLL to output profiler results....297
Draw message numbers -- option....350
Driver Verifier
Problems with....385
Dump folder -- option....175
Dumps -- Generating for profiled applications....482
Duplicated Code -- Profiling of....469
Dynamic link libraries
Profiling of....405
Using custom DLL to output profiler results....297

E

Editor panel....162
Description....162
in RAD Studio....166
in Visual Studio....164
Options....168, 169, 170
Elapsed Time counter....381
Embarcadero
Integration into RAD Studio....16
Preparing C++Builder 2010 applications....86
Preparing Delphi 2010 applications....72
Supported compilers....15
Enable profiling from application code....481
EnableProfiling method....505
End color -- Editor setting....168
Event mark style -- option....187
Event Marks -- option....187
Event source name -- option....176
Event View panel....170
Adding custom messages to the Event View panel....178
Description....170
Exceptions in the Event View Panel....177
Filtering exceptions....178
Options....174
Possible problems with the call stack....179
Event View panel -- option....176
Excel....497
Exporting resuts to....497
Exception Trace profiler....286
Exceptions
Excepitons filter -- option....176
Generating dumps for exceptions automatically....482
Exceptions -- option....176
Excluding code from profiling....125
Explorer panel....180
Description....180
Options....182
Export results....497
ExportCurrentResults method....505
Exporting profiling results to database....510
About....499
Export via COM....510

External DLL name -- option 299
External DLL output -- option.....299

F

F ile names with path -- option and toolbar item.....269
FAQ.....23
File name -- option 176
File names with path -- option and toolbar item.....240, 273, 286, 300, 308, 336, 376
Filter objects by stack -- option and toolbar item 268, 336
Filter standard leaks -- option and toolbar item 268, 336
Filtering exceptions in the Event View panel.....178
Filtering results.....494
Finding 493
  Memory leaks 396
  Values in results.....493
Finding where a method or a class is defined in source code.....389
Five bytes restriction 470
Folder name -- option 183
Font character set -- Editor setting 168
Font name -- Editor setting.....168
Font size -- Editor setting 168
Force Garbage Collection – menu and toolbar item .141
Fortran 118
  Preparing Visual Fortran applications.....118
  Supported versions.....15
Frequently asked questions.....23
Full Analysis -- option value 326
Function Trace profiler.....286
  About Function Trace Profiler 286
  Description of the Details panel columns 295
  Description of the Report panel columns.....291
  Error description on blue screen 385
  Options.....298
  Outputting results using a custom DLL 297
Functions
  Tracing parameters with Function Trace 290

G

Garbage collector routine in profiler results.....213
GCC.....117
  Preparing GNU CC applications.....117
  Supported versions.....15
GDI and User Resources -- option value.....337
GDI+ Resources -- option value.....337
General -- option 174
General settings (Editor settings) 168
Generate dump on exception -- option 175
Generate Process Dump -- toolbar or menu item 482
Generating
  Dumps for profiled applications 482
  Results during testing.....399
Get Results -- menu item 399
Get trace info -- option 299
GetCurrentThreadName function 483
GetParameterName method..... 509
GetParameterValue method..... 509
GetRunMode method 509
Getting results during testing..... 399
Getting support 23
GNU CC 117
  Analyzing GNU CC applications with the Allocation profiler 252
  Preparing GNU CC applications..... 117
  Supported versions..... 15
Grid lines display mode -- option 160
Grid settings -- option..... 160
Grouping results 492
Gutter -- Editor setting..... 168
Gutter capacity -- Editor setting 168

H

Hard Memory Page Faults counter 384
Header part -- option..... 150
Hide IsBadPtr exceptions -- option..... 176
Hide recursion -- option..... 156
Highlight -- option..... 150
Highlight settings (Editor settings)..... 169
How to 387, 398, 462, 487
  Add results as comments to source files 497
  Attach to process..... 402
  Checking bounds of memory blocks..... 254
  Clear results during profiling 401
  Compare results 488
  Controlling profiling from application code 481
  Copy results 497
  Disable inlining for the managed application to be profiled..... 388
  Export results into XML, HTML, XLS or text format 497
  Filter results 494
  Find memory block violations 388
  Find routines exported and imported by a module388
  Find the routine where an exception occurred 389
  Find where a method or a class is defined in source code..... 389
  Get results during testing 399
  Group results..... 492
  Know average maximum and minimum execution times for a method..... 389
  Know if a method raised exceptions 390
  Know on which platforms your application can run 390
  Know parameters and result values of function calls 390
  Know the number of clients that refer to an interface object..... 390
  Know the number of entries into a method..... 391

Know the structure of potential interlinks between classes in your application 391
Know the structure of references between objects in your application 391
Know the structure of routine calls in your application 391
Know the total time spent executing a method (including child methods) 392
Know the total time spent on a method (excluding child methods)....392
Know what binary or MSIL code a method has....392
Know what libraries your application uses 392
Know what methods are called the most or the least often 393
Know what methods take up the most or the least execution time....393
Know what methods use the most time for JIT compiling 394
Know what methods were executed....394
Know what source code lines are called the most or the least often 395
Know what source code lines take up the most or the least execution time 395
Know what source code lines were executed....395
Know which routine allocated a resource 397
Merge results....488
Optimize the profiling process....398
Print profiling results 496
Profile ASP.NET applications 408
Profile ASP.NET applications via ASP.NET Development Server 410
Profile ASP.NET applications via Internet Information Services....413
Profile child routines along with parents 477
Profile COM logical threads 438
Profile COM+ and MTS applications 435
Profile DCOM applications 434
Profile duplicated code 469
Profile dynamic link libraries....405
Profile functions that do not have ret instruction ..476
Profile functions that hold unsafe code....474
Profile IIS applications 421
Profile in-process COM servers....432
Profile managed and unmanaged code....466
Profile multiple processes 462
Profile multiple threads....459
Profile NT services 450
Profile out-of-process COM servers 433
Profile recursive routines 467
Profile scripts 439
Profile small functions 470
Profile startup code 478
Profile system calls 464
Profile template functions 474
Profile Web server applications 407
Profile WPF Browser applications....457
Profile XAML Browser applications 457
Profile XBAP applications 457
Profiler COM applications 431
Search for errors in resource management functions 397
Search for memory leaks.... 396
Search for resource leaks 397
Search results 493
Sort results 492
Trace references between objects.... 398
Tracing attempts to access released memory 253
Tracing system management memory functions.. 253
HTML file -- Exporting results to 497

I

IaqBaseManager interface 502
IaqTimeIntegrationRunMode object.... 509
IaqTimeIntegrationSupportManager interface.... 502
Ignore units with names containing -- option 380
IIS applications 421
IIS applications -- Profiling of 421
Include routine body in Details -- option and toolbar item 240
Incremental search 493
Inline functions -- Profiling of 471
In-process COM servers -- Profiling of 432
Integration.... 16
Borland Developer Studio.... 16
CodeGear RAD Studio.... 16
Embarcadero RAD Studio 16
Microsoft Visual Studio 16
Microsoft Visual Studio Team System 511
Integration assemblies (SQL Server CLR integration) Profiling of 451
IntegrationManager property 502
Intel C++.... 117
Analyzing Intel C++ applications with the Allocation profiler 252
Preparing Intel C++ applications 117
Supported versions.... 15
Internet Information Services 413
ISAPI applications -- Profiling of 421

J

J#49, 52
Preparing J# .NET applications.... 52
Preparing J# 2005 and 2008 applications.... 49
Supported versions.... 15
JIT compiler routine in profiler results 213
JScript profiling
Description.... 439
In Internet Explorer 8.... 448
Prerequisites for profiling 441
Profiling script files.... 445
Profiling scripts located on web pages.... 447
Profiling scripts using host applications.... 445

Troubleshooting 449

K

Keep highlighting when printing -- Editor setting 168
Kernel Resources -- option value 337
Knowing
Average execution time for a method 389
If a method raised exceptions 390
In which routine an exception occurred 389
Maximum execution time for a method 389
Minimum execution time for a method 389
On which platforms your application can run 390
Result values of function calls 390
The number of clients that refer to an interface object 390
The number of entries into a method 391
The structure of potential interlinks between classes in your application 391
The structure of references between objects in your application 391
The structure of routine calls in your application 391
The total time spent executing a method (including child methods) 392
The total time spent on a method (excluding child methods) 392
What binary or MSIL code a method has 392
What libraries your application uses 392
What methods are called the most or the least often 393
What methods take up the most or the least execution time 393
What methods use the most time for JIT compiling 394
What methods were executed 394
What routines are exported and imported by a module 388
What source code lines are called the most or the least often 395
What source code lines take up the most or the least execution time 395
What source code lines were executed 395
Which memory block produced out of bounds operation 388
Which routine allocated a memory block 396
Which routine allocated a resource 397
Which routine created an object 396

L

Leaks
Searching for 396
Light Coverage profiler 300
About Light Coverage profiler 300
Description of results 300
Description of the Details panel columns 308
Description of the Report panel columns 306
Options 308
Line level 126
Line numbers -- Editor setting 168
Load Library Tracer 309
About Load Library Tracer 309
Description of the Details panel columns 314
Description of the Report panel columns 313
Options 315
Loading results from a file 497

M

Manager property 502
Mark partially executed lines as -- option and toolbar item 285, 308
Max consecutive exceptions -- option 176
Maximum number of fields -- option 299
Maximum route depth -- option 299
Memory block bounds – Checking 254
Memory dumps -- Generating for profiled applications 482
Memory leaks
Searching for 396
Merging results 488
Metadata -- How AQtime profilers use it 43
Microsoft C# 45, 48
Preparing C# .NET applications 48
Preparing C# 2005 and 2008 applications for 45
Supported versions 15
Microsoft J# 49, 52
Preparing J# .NET applications 52
Preparing J# 2005 and 2008 applications 49
Supported versions 15
Microsoft Visual Basic 53, 57, 71
Analyzing Visual Basic applications with the Allocation profiler 247
Preparing VB .NET applications 57
Preparing VB 2005 and 2008 applications 53
Preparing VB 6 applications 71
Supported versions 15
Microsoft Visual C++ 58, 61, 68
Analyzing Visual C++ applications with the Allocation profiler 245
Preparing Visual C++ .NET applications 61
Preparing Visual C++ 2005 and 2008 applications 58
Preparing Visual C++ 6 (or earlier) applications ... 68
Microsoft Visual Studio
Integration into 16
Supported versions 15
Microsoft Windows Vista -- Using AQtime under 15
Mixed code 466
Profiling of 466
Monitor panel 183
Description 183
Options 186
Using with the Allocation profiler 187

MTS applications -- Profiling of 435
Multiple processes -- How to profile 462
Multiple threads profiling 459

N

Name property 509
NET Framework assemblies -- Profiling of 464
NET runtime -- Profiling of 240
NewProjectFromModule method 505
Newsgroups about TestComplete 23
Non-executed -- option value 285
Non-existent resources in the Report panel 335
NT event log -- option 176
NT services -- Profiling of 450
Number of child levels -- option 150
Number of parent levels -- option 150
Number of recent results to keep -- option 183

O

Obsolete and Unsupported -- option value 326
OLE -- Working with AQtime via OLE 500
  About working with AQtime via COM 500
  Export results to database via COM 510
  Getting and setting profiling mode parameters 509
  Members of the IntegrationManager class 505
  Working from .NET applications 504
Only Unsupported -- option value 326
Opening a project 120
OpenProject method 505
Optimizing the profiling process 398
Options 155
  Allocation profiler 268
  BDE SQL profiler options 273
  Call Tree panel options 155
  Coverage profiler 285
  Function Trace profiler 298
  Light Coverage profiler options 308
  Load Library Tracer 315
  Performance profiler 239
  Platform Compliance profiler 326
  Reference Count profiler 321
  Resource profiler 336
  Sequence Diagram Link profiler 350
  Static Analysis profiler 376
  Unused VCL Units profiler 380
Out-of-process COM servers -- Profiling of 433
Output -- option 299
Output diagram options -- option 350
Overview of AQtime 10

P

Panel options 155
  Assistant Options 146
  Call Graph Options 149
  Call Tree panel options 155
  Details Options 157
  Disassembler Options 160
  Editor Options 168, 169, 170
  Event View Options 174
  Explorer Options 182
  Monitor Options 186
  Report Options 196
  Setup Options 200
Panels 33, 146, 147, 150, 156, 157, 162, 170, 180, 183, 190, 194, 197, 201
  Assistant 146
  Call Graph 147
  Call Tree 150
  Details 156
  Disassembler 157
  Editor 162
  Event View 170
  Explorer 180
  Monitor 183
  PE Reader 190
  Report 194
  Setup 197
  Summary panel 201
Parameters 290
  Parameters of function calls 290
ParametersCount property 509
Parent item color -- option 150
Partially executed -- option value 285
Path to debug info files
  Specifying 485
PE Reader panel 190
Performance profiler 203
  <JIT compiler> and <Garbage collector> routines 213
  <Root> routine in results 215
  About Performance profiler 203
  Analyzing profiler results 205
  Calculating percent in the Report panel 212
  Description of profiler results 205
  Description of the Details panel columns 224
  Description of the Report panel columns 215
  Error description on blue screen 385
  Options 239
  Searching for bottleneck reasons 240
Platform Compliance profiler 322
  About Platform Compliance profiler 322
  Description of the Report panel columns 325
  Options 326

Print Spooler Resources -- option value 337
Printing profiler results 496
Process dumps -- Generating for profiled applications 482
Processor family -- option 161
Processor models 19
Profile .NET runtime -- otpion 240
Profile <Root> routine -- option 240
Profiler options
  Allocation profiler 268
  BDE SQL profiler 273
  Coverage profiler 285
  Function Trace profiler 298
  Light Coverage profiler 308
  Load Library Tracer 315
  Performance profiler 239
  Platform Compliance profiler 326
  Reference Count profiler 321
  Resource profiler 336
  Sequence Diagram Link profiler 350
  Static Analysis profiler 376
  Unused VCL Units profiler 380
ProfilerCategory property 505
ProfilerName property 505
Profilers 38, 242, 326
  Allocation profiler 242
  BDE SQL profiler 270
  Coverage profiler 274
  Doing one profiler run 139
  Exception Trace profiler 286
  Function Trace profiler 286
  Light Coverage profiler 300
  Load Library Tracer 309
  Performance profiler 203
  Platform Compliance profiler 322
  Reference Count profiler 316
  Resource profiler 326
  Selecting a profiler 137
  Sequence Diagram Link profiler 347
  Static Analysis profiler 350
  Unused VCL Units profiler 376
ProfilersCount method 505
Profiling
  .NET applications -- Peculiarities 486
  5 bytes restriction 470
  64-bit applications 404
  Allocation profiler 242
  ASP.NET applications 408
  Assingning names to threads 483
  Attaching to process 402
  BDE SQL profiler 270
  Child routines along with parents 477
  Clearing results during profiling 401
  COM applications 431
  COM logical threads 438
  COM+ and MTS applications 435
  Controlling profiling from application code 481
  Coverage profiler 274
  DCOM applications 434
  Duplicated code 469
  Dynamic link libraries 405
  Exception Trace profiler 286
  Function Trace profiler 286
  Functions that do not have ret instruction 476
  Functions that hold unsafe code 474
  Getting results during testing 399
  IIS applications 421
  Inline functions 471
  Light Coverage profiler 300
  Load Library Tracer 309
  Managed and unmanaged code 466
  Multiple processes 462
  Multiple threads 459
  Performance profiler 203
  Platform Compliance profiler 322
  Profiling tips 398
  Recursive routines 467
  Reference Count profiler 316
  Resource profiler 326
  Scripts 439
  Sequence Diagram Link profiler 347
  Services 450
  Small functions 470
  SQL Server CLR integration assemblies 451
  Startup code 478
  Static Analysis profiler 350
  System calls 464
  Template functions 474
  Under 64-bit platforms 404
  Unused VCL Units profiler 376
  Web server applications 407
  WPF Browser applications 457
  x64 applications 404
  XAML Browser applications 457
  XBAP applications 457
Profiling areas 124, 127
  Checking elements to profile 131
  Description 127
Profiling levels 126
Profiling modes 401
Profiling results 141
  Export to database 499
Profiling scripts
  Description 439
  In Internet Explorer 8 448
  Prerequisites for profiling 441
  Profiling script files 445
  Profiling scripts located on web pages 447
  Profiling using host applications 445
  Troubleshooting 449

ProfilingStarted property 505

Q

qtdock files 520
Quit method 502

R

RAD Studio
  Integration into 16
  Preparing Delphi 2007 applications 78
  Preparing Delphi 2009 applications 75
  Preparing Delphi 2010 applications 72
  Supported versions 15
Real-Time Monitor 183
Recursive routines -- Profiling of 467
Reference Count profiler 316
  About Reference Count profiler 316
  Description of the Details and Call Tree panel columns 320
  Description of the Report panel columns 319
  Options 321
Registry Resources -- option value 337
Released memory blocks – Tracing with the Allocation profiler 253
RemoveModule method 509
Replays 383
Report panel 194
  Adding routines and classes to the Setup panel 487
  Allocation profiler results 265
  BDE SQL profiler results 272
  Calculating percent 212
  Coverage profiler results 282
  Description 194
  Function Trace profiler results 291
  Light Coverage profiler results 306
  Load Library Tracer results 313
  Options 196
  Performance profiler results 215
  Platform Compliance profiler results 325
  Reference Count profiler results 319
  Resource profiler results 333
  Static Analysis profiler results 372
  Unused VCL Units profiler results 379
Resource categories to check -- option 337
Resource profiler 326
  About Resource profiler 326
  Analyzing profiler results 327
  Description of profiler results 327
  Description of the Details panel columns 335
  Description of the Report panel columns 333
  List of checked functions 337
  Non-existent resources in the report panel 335
  Options 336
Restrictions 470, 474, 476
  Profiling functions that do not have ret instructions 476
  Profiling functions that hold unsafe code 474
  Profiling functions which size is 5 bytes 470
Result views 495
ResultReady property 505
Results 141, 487, 488, 492, 496
  <Garbage collector> routine in profiler results 213
  <JIT compiler> routine in profiler results 213
  <Root> routine 215
  Adding as comments to source files 497
  Adding routines and classes to the Setup panel 487
  Comparison 488
  copying 497
  Duplicated code 469
  Export to database 499
  Exporting 497
  Grouping 492
  Merging 488
  Printing 496
  Recursive routines 467
  Searching 493
  Sorting 492
ret instruction
  Profiling functions that do not have ret instruction 476
Right margin -- Editor setting 168
Right margin position -- Editor setting 168
Root rotuine in profiling results 215
Routine level 126
RunModeCount property 509

S

Saving results to a file 497
Scripts
  About profiling of 439
  Prerequisites 441
  Profiling in Internet Explorer 8 448
  Profiling script files 445
  Profiling scripts located on web pages 447
  Profiling scripts using host applications 445
  Troubleshooting 449
Searching
  Bottlenecks 240
  Performance issues 240
Searching
  For memory leaks 396
Searching
  For resource leaks 397
Searching
  For errors in resource management functions 397
Searching 493
Searching
  Using incremental search 493
Searching
  Values in results 493

SelectProfiler method 505
SelectRunMode method 509
Sequence Diagram Link profiler 347
  About Sequence Diagram Link profiler 347
  Options 350
Services -- Profiling of 450
SetCurrentThreadName function 483
SetParameterValue method 509
Setup panel 197
  Adding routines and classes from the Report panel 487
  Description 197
  Options 200
Show addresses as RVA -- option 161
Show addresses as RVA -- option and toolbar item 376
Show all loaded classes -- option and toolbar item 268
Show all parents -- option 175, 176
Show all resources -- option and toolbar item 336
Show call stack -- option 175
Show critical path for column -- option 156
Show event marks -- option 187
Show footer -- option 157, 196
Show full module path -- option 175, 176
Show group footer -- option 196
Show Group Panel -- menu item 492
Show instruction description -- option 161
Show multiple functions -- option 299
Show non-hit routines -- option and toolbar item 240
Show panel header -- option 161
Show pointing-hand cursor -- option 150
Show results as hint -- option 150
Show results for all profilers -- option 183
Show routine names with class names -- option and toolbar item 273
Show routines with class names -- option and toolbar item 240, 269, 286, 300, 308, 336, 376
Show source line summary -- option 160
Show stack address as RVA -- option 175
Show summary -- option 160
Small functions -- Profiling of 470
Soft Memory Page Faults counter 384
Sorting results 492
Source control systems
  Working with 516
Source line color -- option 161
Source lines -- option 161
Specifying path to debug info files 485
Split Load Replays counter 382
Split Store Replays counter 382
SQL Server integration assemblies
  Profiling of 451
Stack depth -- option 315
Stack Frames - Compiler Settings 179
Start color -- Editor setting 168
Start method 505
Startup code -- Profiling of 478
Static Analysis profiler 350
  About Static Analysis profiler 350
  Analyzing profiler results 362
  Columns of the Details and Call Tree panels 374
  Description of profiler results 362
  Description of the Report panel columns 372
  Options 376
Structure of XML Results 498
Style -- Editor setting 168
Summary panel 201
Support -- Contacting AutomatedQA support team 23
Supported development tools 15
Supported processor models 19
System calls -- Profiling of 464
System management memory functions – Tracing with the Allocation profiler 253
System requirements 13

T

Tab width -- option 161
Tab-delimited text 497
  Exporting results to 497
TakeSnapshot method 505
Target application -- option 350
Technical support 23
Template functions -- Profiling 474
TerminateProfiling method 505
TestComplete information 518
Text file 497
  Exporting resuts to 497
Text file -- option 176
Text file output -- option 299
Text output directory -- option 299
Text separator -- option 299
Thread model -- option 239, 269, 273, 285, 299, 337
Threads 459
  Assigning names to 483
  Profiling 459
Time from application start -- option 175
Timing profiler 203
Tracing attempts to access released memory 253
Tracing references between objects 398
Tracing system management memory functions 253
Triggers 132
  Description 132
  Setting up triggers 133

U

Unsafe code
  Profiling functions that hold unsafe code 474

Unused VCL Units profiler 376
About Unused VCL Units profiler....376
Details panel columns....379
Options....380
Report panel columns 379
Update interval -- option 187
Upper case mnemonics -- option....161
User account
Running .NET applications under....486
User Account....430
ASP.NET 430
IIS 430
User interface overview....25
User Keywords settings (Editor settings) 170
User Time counter....381
User+Kernel Time counter....381

V

VBScript profiling
Description....439
In Internet Explorer 8....448
Prerequisites for profiling 441
Profiling script files 445
Profiling scripts located on web pages....447
Profiling scripts using host applications 445
Troubleshooting....449
View project classes only -- option and toolbar item 268, 336
Visible time range -- option....187
Vista -- Using AQtime under Windows Vista....15
Visual Basic....53, 57, 71
Analyzing Visual Basic applications with the Allocation profiler....247
Preparing VB .NET applications 57
Preparing VB 2005 and 2008 applications....53
Preparing VB 6 applications 71
Supported versions....15
Visual C#....45, 48, 64
Preparing C# .NET applications 48
Preparing C# 2005 and 2008 applications 45
Preparing C#Builder 2006 applications 64
Preparing C#Builder applications 64
Supported versions....15
Visual C++ 58, 61, 68
Analyzing Visual C++ applications with the Allocation profiler.... 245
Preparing Visual C++ .NET applications.... 61
Preparing Visual C++ 2005 and 2008 applications 58
Preparing Visual C++ 6 (or earlier) applications ... 68
Supported versions.... 15
Visual Fortran.... 118
Preparing Visual Fortran applications.... 118
Supported versions.... 15
Visual J#....49, 52
Preparing J# .NET applications.... 52
Preparing J# 2005 and 2008 applications.... 49
Supported versions.... 15
Visual Studio
Integration into.... 16
Supported versions.... 15
Visual Studio Team System – Integration into.... 511

W

Warning level -- option.... 350
Web server applications.... 407
Profiling of.... 407
What is AQtime.... 10
What's New.... 20
Windows DDK
Problems with 385
Using AQtime with 15
Windows Vista -- Using AQtime under 15
Word wrap -- option 161
Working with source control systems.... 516
WPF Browser applications -- Profiling of 457

X

x64 applications -- Profiling of.... 404
x86 and x64 AQtime Packages.... 13
x86 Disassembler -- option.... 161
XBAP applications -- Profiling of 457
XLS file -- Exporting results to 497
XML file....497, 498
Exporting results to 497
Structure of XML results 498
XML file -- option 176